1984

Remote terminal login from a microcomputer to the UNIX operating system using Ethernet as the communications medium.

Reeke, John Donald

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REMOTE TERMINAL LOGIN FROM A MICROCOMPUTER TO THE UNIX OPERATING SYSTEM USING ETHERNET AS THE COMMUNICATIONS MEDIUM

by

John Donald Reeke
December 1984

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Remote Terminal Login From A Microcomputer To The UNIX Operating System Using Ethernet As The Communications Medium		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Donald Reeke		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE Master's Thesis
		13. NUMBER OF PAGES 121
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

UNIX, Ethernet, Remote Login

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis examines the viability of a microcomputer based process which allows its terminal to become a terminal to a remote host which utilizes the UNIX operating system. The means of communication to the remote host is an Ethernet local area network.

The preponderance of the software to accomplish the remote login is PL/I thus enabling the software to be utilized (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

by any combination of computer/operating system that supports PL/I. The Ethernet driver makes use of assembly language in order to program the Ethernet Controller Board and is therefore dependent upon the underlying microprocessor.
Although the communications medium is the Ethernet, ARPANET Internet and Transmission Control protocols are also utilized. This is necessitated by the manner in which the UNIX operating system's remote login process operates and allows the far more general case where the remote host is part of the ARPANET.

Approved for public release; distribution is unlimited

Remote Terminal Login from a Microcomputer to the UNIX Operating System Using Ethernet as the Communications Medium

by

John Donald Reeke
Lieutenant Colonel, United States Marine Corps
B.S., Regis College, 1966

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

This thesis examines the viability of a microcomputer based process which allows its terminal to become a terminal to a remote host which utilizes the UNIX operating system. The means of communication to the remote host is an Ethernet local area network.

The preponderance of the software to accomplish the remote login is PL/I thus enabling the software to be utilized by any combination of computer/operating system that supports PL/I. The Ethernet driver makes use of assembly language in order to program the Ethernet Controller Board and is therefore dependent upon the underlying microprocessor.

Although the communications medium is the Ethernet, ARPANET Internet and Transmission Control protocols are also utilized. This is necessitated by the manner in which the UNIX operating system's remote login process operates and allows the far more general case where the remote host is part of the ARPANET.

DISCLAIMER

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below following the firm holding the trademark:

Digital Research Incorporated, Pacific Grove, California

CP/M-86 Operating System

PL/I-86 Programming Language

Link-86 Linking Utility

DDT-86 Dynamic Debugging Tool

Intel Corporation, Santa Clara, California

86/12A Single Board Computer

Multibus Architecture

Digital Equipment Corporation, Maynard, Massachusetts

VAX 11/780 Minicomputer

VMS Operating System

Interlan Corporation, Chelmsford, Massachusetts

NI3010 Ethernet Controller Board

Xerox Corporation, Stamford, Connecticut

Ethernet Local Area Network

Sun Microsystems Incorporated, Mountain View, Ca.

Sun Workstation

Bell Laboratories, Murray Hill, New Jersey

UNIX Operating System

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

AEGIS is a highly sophisticated weapons system designed to safeguard a surface fleet against enemy aircraft and missiles. An integral part of the system is the SPY-1A radar; it feeds its data to a Navy standard AN/YUK-7 computer.

The AEGIS weapons system simulation project is an ongoing study being conducted at the Naval Postgraduate School. Its primary objective is to study the feasibility of replacing the present four-processor AN/YUK-7 mainframe computer with a multi-microcomputer based architecture.

The AEGIS weapon system must process in real-time large amounts of data concerning target detection and acquisition. In order to distribute the data among the many micro-computers, high speed data communication means are needed. The high speed data transfer can be accomplished by tying clusters of MULTIBUS connected micro-computers to each other by means of a local area network such as Ethernet. The 10 megabit per second data transfer rate over a maximum of 2500 meters of cable makes Ethernet a powerful data communications medium which replaces a complex system of point to point cables with a single cable.

In order to create a useful program development environment for the AEGIS Modeling Project, it is convenient to access more powerful processors located outside of the micro-computer cluster but connected to the same Ethernet. The UNIX operating system has been popular on mini-computers and is becoming much more commonplace on the newer, more powerful micro-computers and is thus a natural candidate for a remote login from a terminal connected to the micro-computer cluster.

Initial groundwork in the AEGIS project for the connection of micro-computers to more powerful processors using the Ethernet was conducted by Stotzer [Ref. 1] and Netniyom [Ref. 2]. Stotzer, working at the micro-processor end, and Netniyom, working at the other end with a VAX 11/780 VMS operating system, implemented message passing and file transfer routines. As VMS did not provide for a remote login as an integral part of the operating system, the communication terminated at the VAX end in one of the device files, thus requiring one "hard wired" terminal from the system. A hoped-for virtual communication therefore became in essence a hard wired connection over the Ethernet.

B. GENERAL DISCUSSION

The Berkeley 4.2 version of the UNIX operating system (4.2 BSD) in use at the Naval Postgraduate School runs on a VAX 11/780 which is connected to an Ethernet. Since this

version of UNIX implements networking software facilities, device drivers for networking hardware, and a remote login capability, the feasibility of remote terminal login over Ethernet to a more powerful remote host from a micro-computer was investigated. The extension beyond Stotzer's and Netniyom's work was important since many users could remotely use the VAX 11/780 over just one communication medium, Ethernet, without using any of the hardwired terminal connections to the remote host. The method of investigation used was to capture for analysis the Ethernet communications between a Sun Workstation, running 4.2 BSD UNIX, and a VAX 11/780, also running 4.2 BSD UNIX.

C. STRUCTURE OF THE THESIS

Chapter I presents some background information on the AEGIS project here at the Naval Postgraduate School and prior work to facilitate micro-computer communication with more capable processors over an Ethernet connection. It also discusses the continuing effort to enhance such communication by being able to remotely become a terminal to a VAX 11/780 from a micro-computer.

Chapter II will discuss, in a rather general manner, the topics of networks, the ARPANET network, and the relation between the ISO standards for networks and the manner in which the ARPANET is implemented. Information of local area networks and Ethernet will also be presented. Some

background knowledge of these areas is necessary since the 4.2 BSD UNIX system implementation of the remote login allows logging in over a long haul network and is thus more versatile and flexible than if it were implemented for logging in over only an Ethernet.

Chapter III covers specific information on the MULTIBUS system, Ethernet controller boards, and the single board computers that were utilized in the course of this investigation.

Chapter IV will discuss the manner in which the two UNIX based computers communicate over the Ethernet when remotely logging in. It also covers in detail the ARPANET protocols: Internet Protocol (IP) and Transmission Control Protocol(TCP), the UNIX login, terminal, and trailer protocols.

Chapter V will discuss the purpose of and results obtained by the demonstration program. It will also discuss several important anomalies that must be avoided should there be work continuing from this thesis.

Chapter VI presents the conclusions of the thesis.

II. NETWORKS

A. NETWORKS: GENERAL DISCUSSION

1. Background

The 4.2 BSD UNIX implements the remote login in a manner that is general enough to enable one to login to a remote host computer over a local area network (LAN), a long haul network such as the ARPANET, or any combination of the two. Thus it would be possible to remotely login from one UNIX system to another by using a local area network to a node located on a long haul network, thence over the long haul network to a remote host that is also running 4.2 BSD UNIX. In such a case the remote login request would be surrounded by both LAN protocols and long haul network protocols, specifically ARPANET protocols. In order to analyze the remote login conversation between two UNIX systems, some basic knowledge of networks is necessary.

2. Definition

A computer network is an interconnection of autonomous computers. The interconnection can be accomplished by a variety of means ranging from hardwired conections, micro-wave connections, or even satellite connections; in fact, some large networks might employ all methods of connections at one link or another within the network. The computers should be thought of as autonomous

in that there is no master/slave relationship between any of the computers on the network.

3. Aims of a Network

The growth of networks over the last decade has been spectacular. This drift away from large, centralized computer systems is caused by the many economic and performance advantages of networking. By using a network, all resources (data, programs, processors, etc.) on the network are potentially available to anyone using the network regardless of the physical proximity of those desired resources. Computing systems can be made much more reliable with the utilization of networks since, should one part of the system break down, other parts can take up the load. Thus systems can degrade in capability gracefully rather than catastrophically. Since the cost of computers has declined dramatically while the cost of communication has remained stable, it now makes more sense to do much more processing of data locally and then communicating only the results to a central location rather than sending all data to a central point for all processing needs. In essence, the tremendous needs generated by the information age are more easily fulfilled by the utilization of networks.

4. Networking Terms

Some definitions of basic networking terms are needed before continuing the discussion. Hosts are the various machines located on the network which are intended

for running user (application) programs. The specialized computers that perform the switching necessary in a network are called IMP's, interface message processors. A link is the physical communication medium used to pass data from one IMP to the next IMP in the network. A typical session on a network might have the user wanting to send a message to another person in a distant location. The user would sit down at his host computer and generate his message which would be communicated to the IMP which actually puts the message onto the network link to the destination. The users data, assuming it is a short message, would be transmitted across the network in a packet, the basic unit of information that moves over the network. A packet switched network is one in which the packet can very greatly in size and a source to destination path need not be established before the packet is sent on its way. For instance, two packets from the same host to the same destination may leave the source host one right after the other but take a different route to the destination host. The first packet transmitted from the source host may not be the first to arrive at the destination.

5. Network Layers

Networks, especially long haul networks, are very complex and require elaborate software to make them run effectively. In order to allow for rapidly changing technology and protocols, the design of software must be

very modular in nature. Also, the explosive growth of networks has made the adoption of standards by all involved of paramount importance. An effort to simplify production of and changes to neworking software by standardizing the modularity of the software is found in the seven layered approach as espoused by the International Standards Organization (ISO). Figure 2.1 shows the ISO standard layer approach to communication from one node to another over a network.

Network communication commences when a user at one node interfaces with the top layer, the application layer. This layer then communicates with the same layer in the destination node by what is called a peer to peer protocol. A protocol is a communication convention between equivalent layers in the source and destination hosts. This is a virtual communication; actually the application layer interfaces with and makes use of the services provided by the next lower layer, the presentation layer. This layer communicates to the presentation layer in the destination host by means of the presentation layer peer to peer protocol. As before, this is a virtual connection; the actual communication by the presention layer is to the next lower layer. The actual communication continues down through the layers until it reaches layer 1, the physical layer. The data to be transferred ends up at the lowest level surrounded by various extra bits appended by each

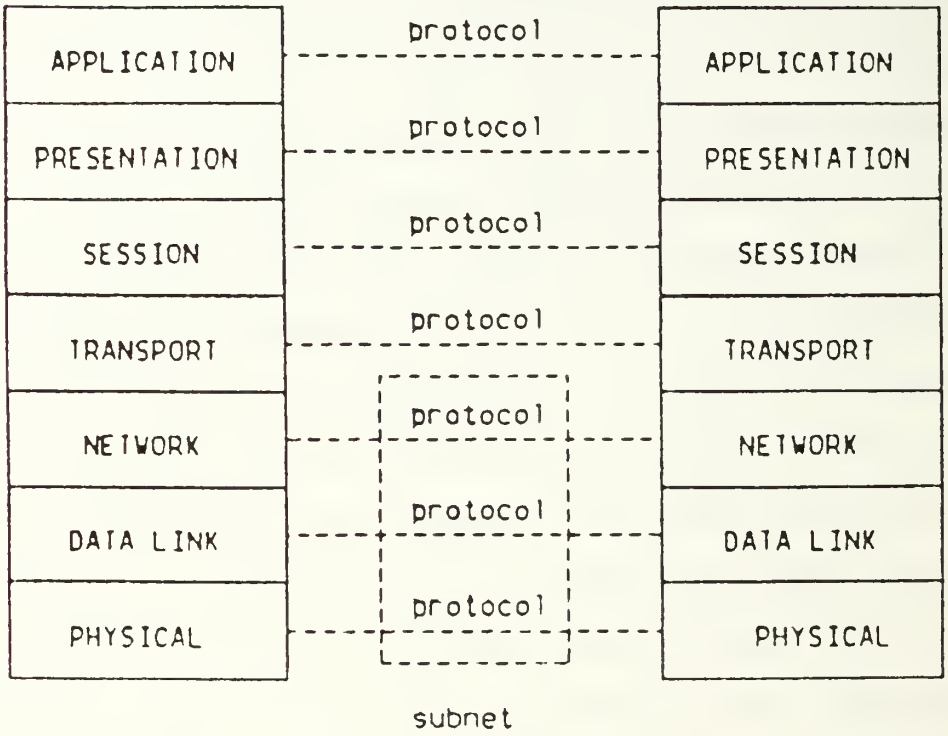

Figure 2.1 ISO Network Layers

layer as the data passes through it. It is from this layer that the data is actually moved to another node on the network by some physical link, hardwire, microwave, etc.. At the destination node, the data ascends through the layers, losing the bits added by the source node layers, until it reaches the destination application layer. Each layer can be pictured as requesting services from the layer beneath it and providing services for the layer above it.

The importance of this layered concept is that functions associated with a network connection can be grouped together in independent software modules that handle only the functions of one particular layer. Should the peer to peer protocols change, then only the one software module associated with that layer need be altered.

6. Functions of the Layers

The first of the seven ISO layers is the **physical layer**. Its function is the transmitting of raw bits of data over a communication channel. The protocol at this layer must ensure that if the source transmits a binary 1, then a binary 1 is received at the destination. It accomplishes this by handling electrical voltages and electrical pulses, cables, and connectors. The Ethernet physical layer uses the protocols known as Carrier Sense Multiple Acces(CSMA)/ Collision Detection (CD).

The function of the **data link layer** is to take the raw transmission provided by the physical layer and transform it

into a line that appears, to higher protocols, to be free of transmission errors. This layer adds flags to denote the beginning and ending of messages, utilizes error checking algorithms are utilized, and distinguishes data from flags. Also, this layer deals with the problems of acknowledgements and of flow control (a fast sender swamping a slow receiver).

The function of the **network layer** is to control the operation of the subnet, the lower three layers. It determines the routing to the destination, prevents congestion, and breaks large packets into smaller ones if they are too large for a particular link, and handles flow control within the network. The network layer may provide either datagram service or virtual circuit service. Datagram service delivers to the next higher layer the messages from the transport layer without ordering the messages; each is considered an isolated unit. The virtual circuit service presents the transport layer with ordered messages. For commercial networks, this layer contains the accounting algorithms for the billing of customers.

The function of the **transport layer** is to ensure host to host error detection and recovery, control the size of messages passed to the next lower level (some long messages might have to be broken up into shorter length messages), multiplex the end user address onto the network, and the establishment/release of connections across the network. It

also routes incoming messages to the correct process or connections within the host and reassembles messages by delivering to the next higher level the parts of messages in the correct order, which is not necessarily the order in which the messages were received.

The **session layer** provides the users interface in the network. It provides the establishment of a connection with a process on another machine. Once the connection has been established, it controls the buffering of data, the data transfer, and the graceful or abrupt closure of the connection.

The function of the **presentation layer** is to negotiate the syntax for characters as data, text strings, terminal display formats, and so on. This layer would also provide for data encryption should that be necessary.

The function of the **application layer** is largely determined by the user. It can provide for such functions as login procedures and passwords. Here the type of service to the user is defined; i.e., is the transaction a file transfer, data transfer, remote login, financial transaction, etc..

B. THE ARPANET

1. Background

The Department of Defense started the funding of research into computer networks during the 1960's. Through

the Defense Advanced Research Projects Agency, money was provided to selected universities and private firms to set up an experimental four node network. It became operational in 1969. It has since expanded to over 100 nodes that span the globe. Because of the length of time it has been in existence, the monies provided for continuing research, and the sponsorship by a powerful backer, the ARPANET is the most significant network in the world. Much of the current day knowledge about networks can be traced to the ARPANET development.

2. ARPANET Layers

The networking layers of the ARPANET do not correspond in a one-to-one manner with those of the ISO standards. Figure 2.2 shows the ARPANET layers and their relation to the ISO standard layers. The ARPANET protocols are grouped into four major categories: network backbone, network access, host to host, and application or service protocols. Each of these is independent and optimized for its special function and may contain several protocols embedded.

The **network backbone protocols** are the IMP-IMP protocols, routing protocols, and end-end protocols. **network access protocols** define the interface between a host computer and the switching nodes(IMP's).

The above protocols are interface protocols between IMP's and the host computer. They permit reliable data

	ISO MODEL	ARPANET MODEL
7	APPLICATION	USER
6	PRESENTATION	TELNET, FTP
5	SESSION	(NONE)
4	TEANSPORT	TCP
3	NETWORK	INTERNET
2	DATA LINK	IMP-IMP
1	PHYSICAL	PHYSICAL

Figure 2.2 ISO Versus ARPANET Layers

transfer into or out of the IMP to the host link or subnetwork. These two protocols do not, however, provide reliable host to host peer communication across the network. This all important function is provided by the ARPANET internet protocol and transmission control protocol, IP/TCP. The protocols will be discussed in depth later in this thesis; they form the major part of the communications format that takes place in the remote login facility of 4.2 BSD UNIX. A less robust host to host protocol can be formed by combining the same internet protocol with the User Datagram Protocol, UDP. This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. It is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications that require an ordered delivery of streams of data should use the TCP vice the UDP protocol.

Above these layers in the ARPANET model are the application or service layers and protocols such as telnet (the virtual terminal protocol), ftp (file transfer protocol), sftp (simple file transfer protocol), etc.. Figure 2.3 shows the ARPANET protocol relationships.

C. LOCAL AREA NETWORKS

1. Definition

A local area network is best defined by listing some of the requirements that are generally accepted as

being necessary in a local area network. It is a collection of automonous computers and other resources that:

a. geographically are within about 2500 meters of one another;

b. have a relatively high bandwidth, usually ranging from 1 to 10 million bits per second;

c. low cost;

d. capability to support up to several hundred connections of devices on the network;

e. low error rates, high reliability, and independence from a centralized control;

f. ease of installation and extensibility;

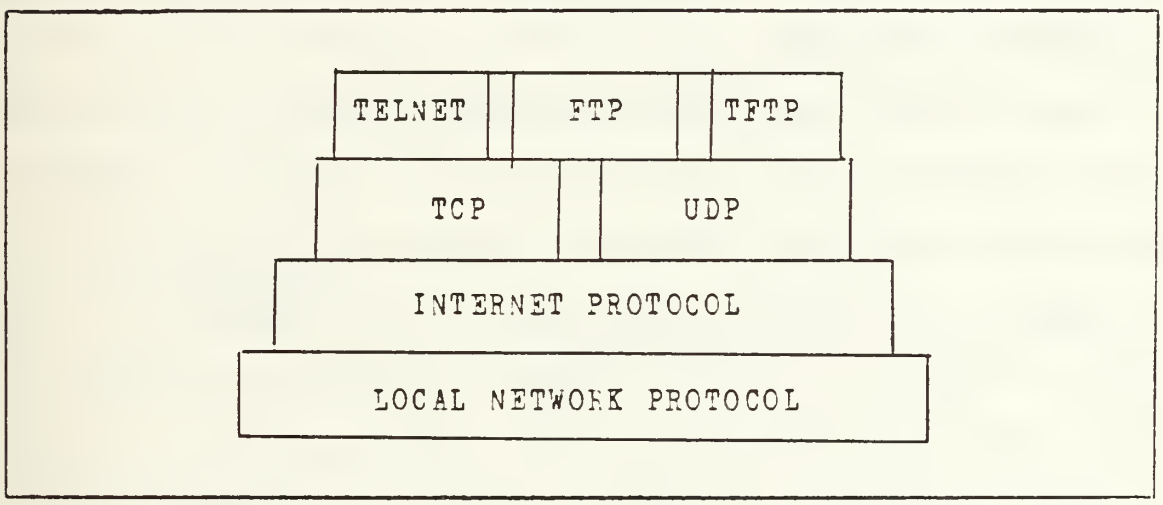

Figure 2.3 ARPANET Protocol Hierarchy

2. Why Local Area Networks

There are several reasons for the high interest in local area networks. Organizations today may have many computers and peripherals located within the same or adjacent buildings. Cost and performance considerations

demand that most if not all of these devices be tied together over some communication medium. The most efficient manner of interconnection, rather than a maze of "hardwired" connections, is a local area network utilizing a high bandwidth physical cable. Each computer or peripheral is then attached to the channel over a very short physical connection. a distinct advantage of a local area network is its ability to more effectively and efficiently perform distributed computing tasks. This allows specific functions such as file storage, data base management, terminal handling and so on to be assigned to specific machines thus simplifying implementation.

Local area networks have several factors that make their implementation much easier than long haul networks. The routing algorithms can be as simple as broadcasting all messages. The routing algorithms on long haul networks can become extremely complex, especially when one network is tied into other networks. Local area networks do have potential congestion problems, but they are minor compared to those that can be encountered on long haul networks.

Local area networks can sense congestion because they sense the state of the broadcast channel before attempting to use it. Two of the types of local area networks are the carrier sense networks and the ring networks; each type has a unique method of sensing the state of the network. The carrier sense method, when it has data to send. listens to

the channel to see if anyone else is broadcasting. If the channel is busy, the station waits until the channel becomes idle. Once it has sensed an idle channel, it transmits all or part of its data in a packet. If a collision occurs with another unsensed packet, thus corrupting the transmission, the station waits a random amount of time and then starts the process over again. The ring networks, like the carrier sense networks, use twisted cable, coaxial cable, or fiber optics as the transmission medium; however, the ring net is a series of point to point cables between consecutive stations. The basic method of the control of network congestion is through a token system. Each node wishing to transmit must have a ticket or token in order to use the network.

3. Ethernet

Ethernet is a local area network specification developed by Digital Equipment Corporation, Xerox Corporation, and Intel Corporation. It is designed for the high speed data exchange among computers and other digital devices located within a moderately sized geographical area. The Ethernet specification pertains to the first two layers of the ISO model, the physical and data link layer. It provides for a data transmission rate of 10 million bits per second over a coaxial cable with a maximum length of 2.8 kilometers. It supports up to 1024 stations with a topology

of a branching non-rooted tree. The link control is CSMA/CD.

4. Ethernet Frame Format

a. General

The Ethernet frame consists of the following five fields: the destination address, the source address, the type field, the data, and finally the frame check sequence. The 64 bit preamble used for receiver synchronization is not available to the user.

b. Destination address

The six byte destination can be one of three types of addresses. It can be the physical address of a board; this address is unique to each board interfacing on the Ethernet; its first three bytes are assigned by Xerox and the last three by the board manufacturer. It could also be a multicast address; this board can be user programmed to contain up to 64 different multicast addresses. All boards receive broadcast packets.

c. Source address

The physical source of the particular board is automatically inserted upon packet transmission. This can be circumvented, as will be explained later.

d. Type field

The two byte type field contents are not specified; the user can use it for whatever purpose he desires.

e. Data field

The data field must be a minimum of 46 bytes and can be as long as 1500 bytes. Should the user data be less than 46 bytes, the unused bytes are stuffed with the null byte.

f. Frame check sequence

This is a 32 bit cyclic redundancy check of the Ethernet frame from the destination address down to the last byte of data.

III. NPS FACILITIES

A. BACKGROUND

The system used at the Naval Postgraduate School to investigate the viability of the remote login over Ethernet was composed of the following components: the AEGIS Project multi-user system, a Sun Workstation and a VAX 11/780, both running 4.2 BDS UNIX. All components of the system were connected by an Ethernet. Figure 3.1 shows an overview of the system. The makeup of the AEGIS multi-user system is shown in Figure 3.2. Its MULTIBUS backplane contains four Intel 86/12 Single Board Computers, two additional memory boards, and an Interlan Ethernet Controller Board. Each 86/12 SBC is attached to its own terminal. Secondary storage for the system is provided by two hard disk drives and two floppy disk drives. Some basic knowledge of the multi-user system components is necessary for the understanding of the programs contained in this thesis.

B. MULTI-USER SYSTEM COMPONENTS

1. Intel 86/12 Single Board Computer

The 86/12 SBC contains all the components necessary for a computer on a single board; it contains an 8086 16-bit microprocessor, 64K bytes of onboard RAM, a serial communications interface, a programmable interrupt controller, and a programmable timer. It also contains the

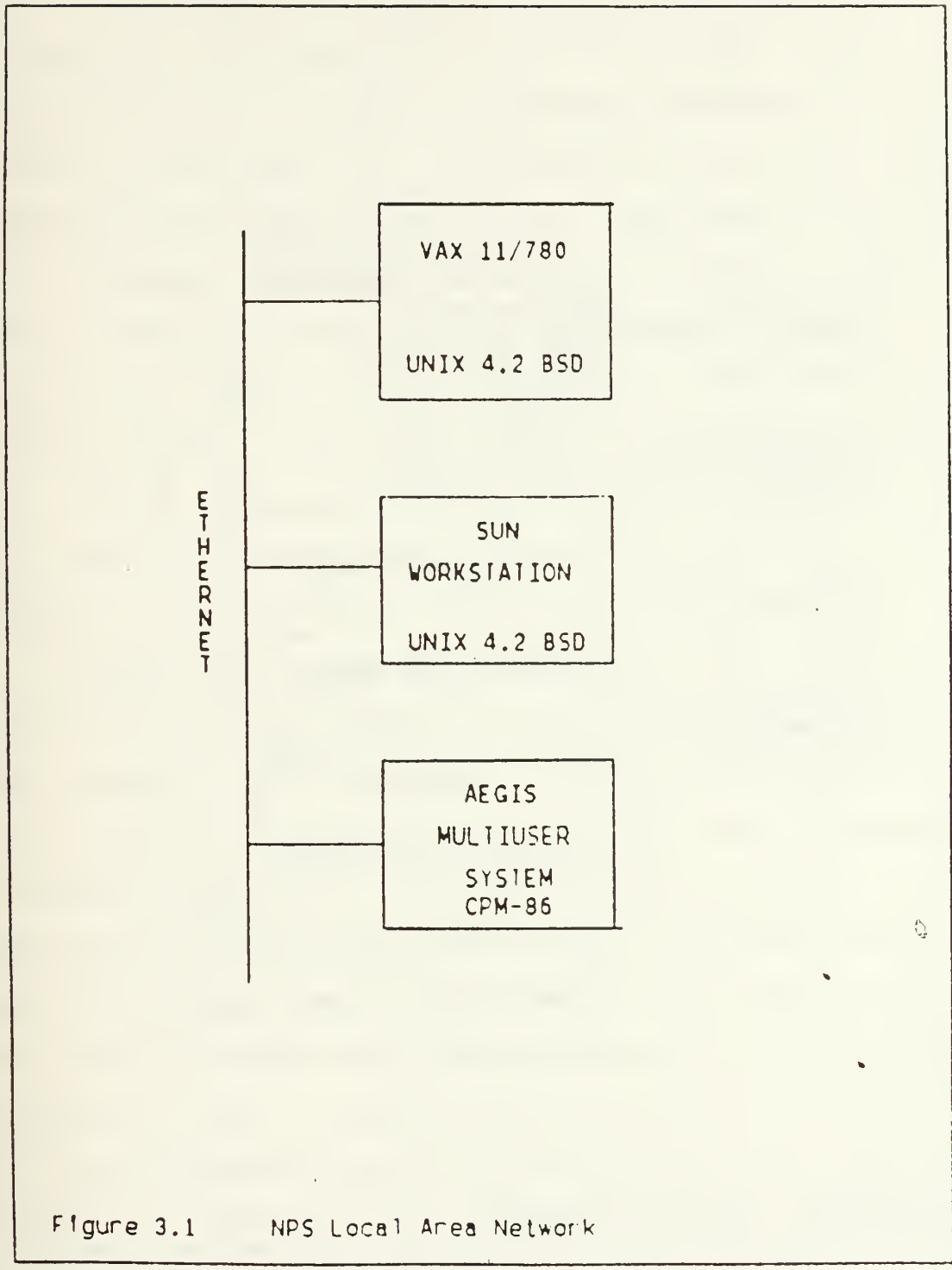

Figure 3.1 NPS Local Area Network

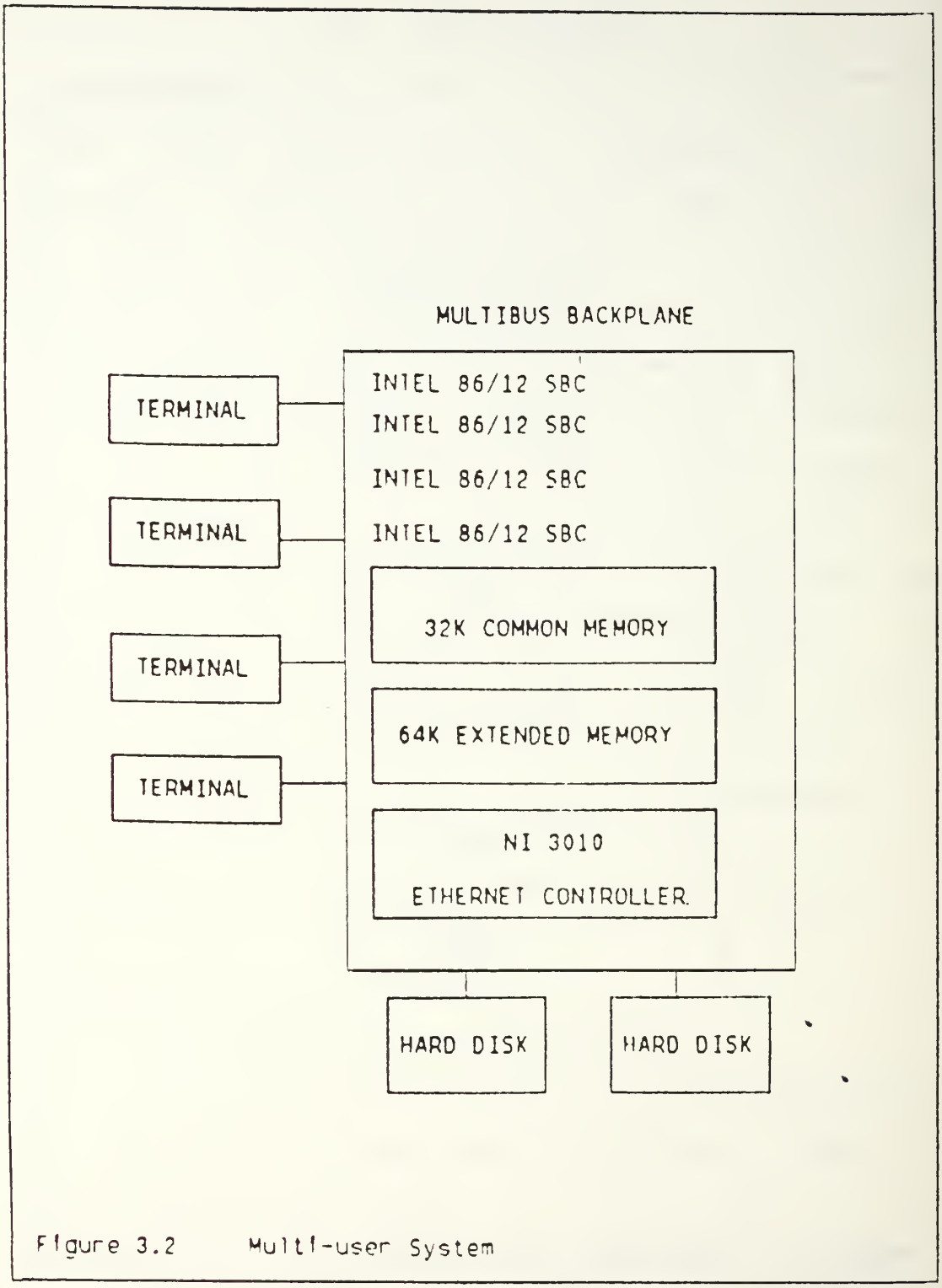

Figure 3.2 Multi-user System

circuitry necessary for the use of the board on a MULTIBUS system. The board is configured so that it can act either as a MULTIBUS slave board or as a master board and can accept user provided ROM of up to 32K bytes. All of the onboard RAM is accessible to offboard components if the board is unstrapped; or, conversely, none of the onboard memory can be accessed from offboard components if the board is strapped. The ability to allow or disallow access to onboard memory (strapping) is a convienent feature of the SBC that makes it very useful in multi-processing installations. In the AEGIS system, the boards are strapped to ensure that no process running on another board accidentally contaminates the onboard memory of another board.

The 8086 was one of the first commercially available 16-bit microprocessors. Figure 3.3 shows the internal structure of the 8086. The processor is divided into an execution unit, EU, and a bus interface unit, BIU. The EU implements a standard single-bus, accumulator based arhitecture and the BIU manages all bus communications to the outside world. The data and addresses are multi-plexed over the same 16 pins of the 40 pin chip. The BIU also implements an instruction pipeline in order to speed up the execution in the 8086. Up to six pre-fetched instructions are queued for execution. General use registers, the AX, BX, CX, and DX registers, are 16 bits wide and are either 8 or 16 bit addressable. There

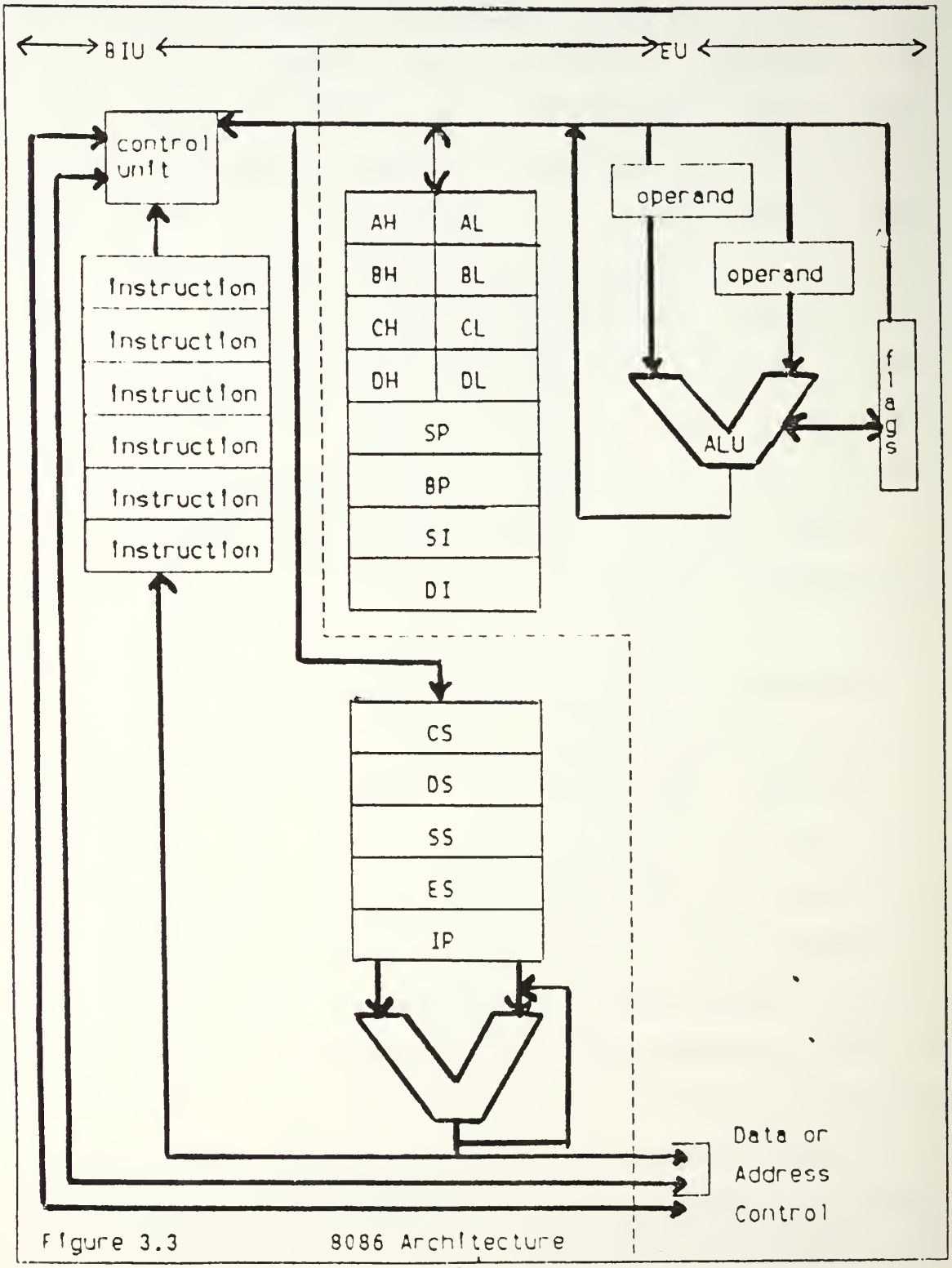

Figure 3.3 8086 Architecture

are also four word addressable registers for indicies and pointers, the CS(code segment), the DS(data segment), the SS(stack segment), and the ES(extra segment). These segment registers serve as base addresses for addressing 64K byte segments within the 20 bit, one megabyte address space of the 8086. In order to effectively address one megabyte, the base address, only 16 bits, is left shifted by four bits so that when added to the 16 bit offset, the full one megabyte address space is available. These segment registers are programmable so that any portion of the one megabyte address space is available. This enables the addressing of memory space which lies outside the 64K of 86/12 SBC onboard RAM. This is crucial for the ability to address the memory expansion boards and other offboard components as part of a process running on the 86/12 SBC.

2. Memory Configuration on the MULTIBUS System

The AEGIS project makes use of a multi user system to run a multiprocessing operating system, MCORTEX, which is appended to the CPM-86 operating system that is resident on each 86/12 SBC. MCORTEX is loaded from disk directly to onboard memory. CPM-86 has a master and a slave version. The master version is loaded from disk to the MULTIBUS master 86/12 SBC. The slave version is loaded onto a memory expansion board called common memory; copies of this are then loaded into slave 86/12 SBC's as needed. This common memory belongs to all boards and processes and is the area

where MCORTEX places various process synchronization data. The other memory expansion board, call it the extended TPA(transient program area), is available to any 86/12 SBC. For the purposes of this thesis, this extended TPA was used by the PL/I programs as a portion of the data segment from and to which the Direct Memory Access operations for the Ethernet Control Board were conducted. Figure 3.4 depicts in a linear manner the memory utilization in the AEGIS multi-user system.

3. Interlan NI3010 Ethernet Controller Board

a. General

The Ethernet controller Board is a MULTIBUS Ethernet communications controller that together with a transceiver implements layers one and two of the ISO network layer. It implements fully a protocol that complies with the Xerox/Intel/Digital Ethernet specification, version 1.0.

The board performs data link layer functions by formatting user data into frames and performing the CSMA/CD sensing. It automatically provides the cyclic redundancy code generation (CRC code). The board, when listening to the network, recognizes and accepts frames which are addressed to the board's physical address, one of its multi-cast addresses, or any broadcast frame.

The controller board performs the physical layer functions by transmitting/receiving bit streams at 10 megabytes per second. It also contains a 16K receive buffer

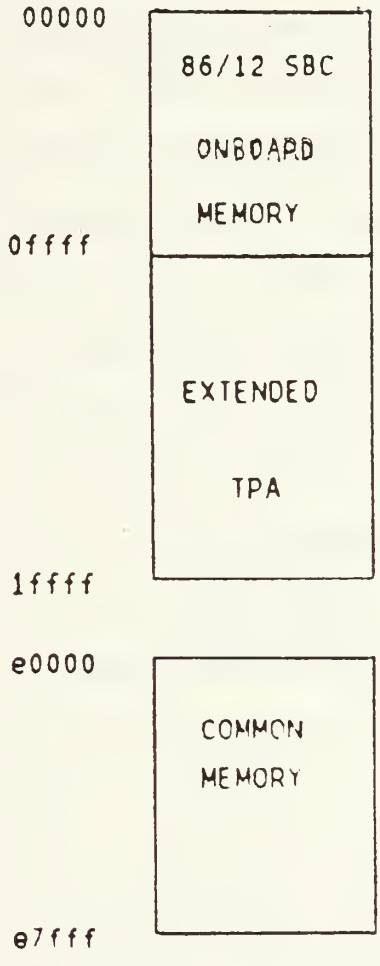

Figure 3.4 Address Space

so that the host computer can read the incoming frames or packets at its own convenience. By buffering, it also shields the MULTIBUS from the unpredictable arrival time of frames. The NI3010 board has the DMA controller that moves packets to/from host memory. The board has control of the MULTIBUS during this transfer. Should another board take over the MULTIBUS between data transfers, the NI3010 board will resume its DMA transfer from the point where it surrendered the bus. The board also maintains a small buffer 2K in length for a storage of data to be transmitted. In the programs included herein, the NI3010 performs DMA transfers to/from the extended TPA memory board. The NI3010 board also performs extensive self-diagnostics and can keep statistics on the network traffic and errors.

b. Frame transmission

The host computer sets up in contiguous memory a transmit data area in the extended TPA. When instructed by the host, the NI3010 board commences a DMA transfer of this data into its transmit buffer. The board does not attempt transmission onto the network until specifically told to do so by the host. Once instructed to do so, it inserts the source address of the board, computes and appends the CRC, and transmits the packet. Should a collision occur that prevents transmission, the board will automatically attempt to retransmit up to 16 times. Figure 3.5 shows the format of the transmit data area.

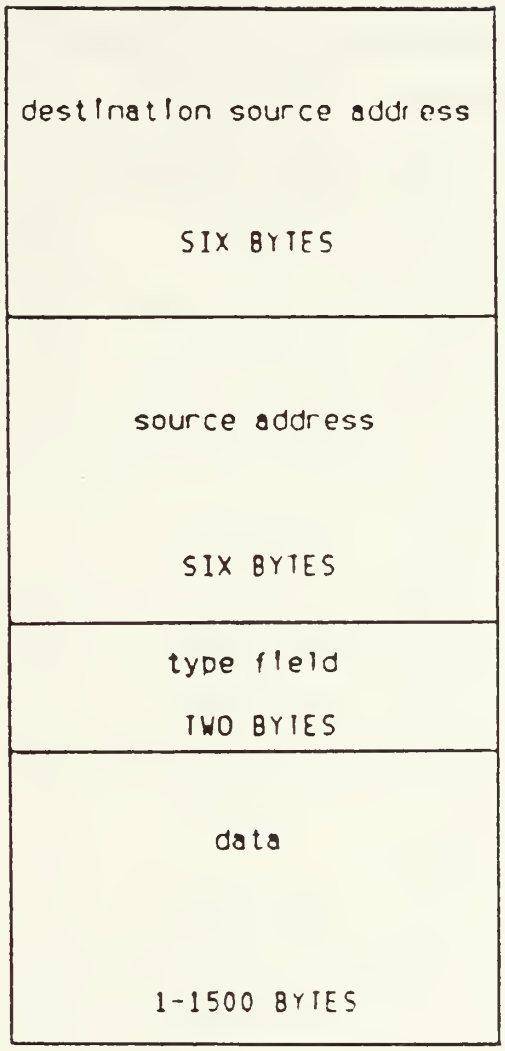

Figure 3.5 Transmit Data Block

c. Frame receptions

When the NI3010 receives a packet that is destined for its host, it informs the host via a hardware interrupt. When the host decides it is ready for the packet, it signals the board which then prefixes frame status and frame length information to the received frame, and then performs a DMA transfer of the packet into a host determined area of memory. In the AEGIS project, it is transferred to the extended TPA portion of memory. Figure 3.6 shows the format of a packet that has been transferred to the receive data area.

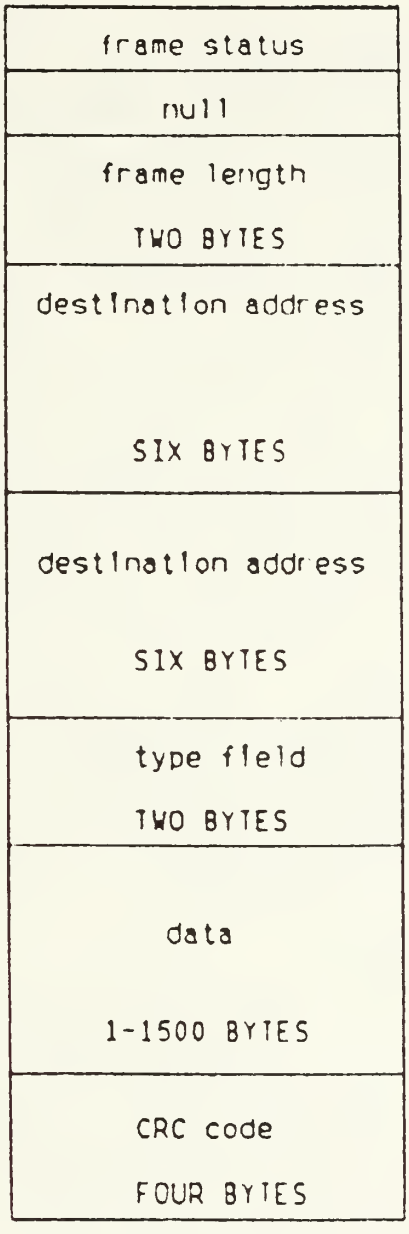

Figure 3.6 Receive Data Block

IV. REMOTE LOGIN PROTOCOLS

A. UNIX REMOTE LOGIN

1. What is a Remote Login

When a user desires to remotely log into the VAX from the Sun Worksation, he executes the UNIX command RLOGIN. The Sun then sends/receives packets over the Ethernet to/from the VAX login port. The terminal attached to the Sun appears to the user as if it were hardwired directly into the VAX. All of the networking implementations are transparent to the user. All of the normal login messages and terminal queries appear, as does the same command line prompt used on the VAX UNIX. All special control characters, such as "kill", "delete", etc., are exactly as those used on the VAX. Investigating the viability of duplicating this process on the AEGIS multiuser system is the goal of this thesis.

2. Packet Contents

The first step taken in investigating the UNIX remote login processs was to produce a program for the AEGIS project's multiuser system that passively listened in on the Ethernet and captured the conversation between the VAX and the Sun when remotely logging in and conducting a terminal session. The program, LISTEN, captured all traffic on the Ethernet and printed it into a file in either hexadecimal or binary format. Since the Ethernet cable had

only three devices connected to it at the time, it was possible to eliminate all traffic on the cable except those packets that were directly involved with this remote login/terminal process.

Examination of the packets revealed that several protocols were present. The most obvious protocol was the Ethernet protocol. The only item of interest here as far as the remote login was the Ethernet header type fields. As mentioned in Chapter II, this field is undefined in the Ethernet specifications and is available to the user for whatever use he deems appropriate. In this case, the low order byte of the type field was always 8 hexadecimal and the high order byte was Ø. The reason for this is not known to the author.

There were two ARPANET protocols present, the Internet and TCP protocols. There were also UNIX login, terminal, and trailer protocols. The login and terminal protocols are very similar to those used during a normal login session when a terminal is hardwired into the VAX UNIX. The conversation over the Ethernet can be viewed as protocols wrapped in protocols. The Ethernet protocols are the outermost layer, then Internet and TCP, then the UNIX protocols.

At first glance it appears that there is a lot of excess baggage used to merely become a terminal over the Ethernet. However, the implementation used by 4.2 BSD UNIX is

sufficiently general so that the remote login could take place from the Sun, over the Ethernet to the VAX, and then over the ARPANET to another node running 4.2 BSD UNIX. A much less complex method could have been implemented, but it would not have complied with ARPANET specifications. Besides, since there is a large bandwidth available on the Ethernet, these extra bytes take up little "space".

B. INTERNET PROTOCOL

1. Background

The Internet Protocol (IP) was developed for use in packet switched, interconnected networks. The fundamental purpose of IP is to transfer data from one host to another over a long haul network and to allow long blocks of data to be broken up into smaller pieces (fragmentation) should a particular link in the network not be able to handle that sized block of data. The block of data transferred by IP is referred to as a datagram. The IP datagram consists of two elements, a header and the actual data. The scope of this protocol is limited; it does not provide for end to end reliability, flow control, sequencing, or other host to host protocols. These later functions are the objects of the next higher protocol, TCP.

Figure 2.2 shows the relation of IP to the other ARPANET protocols. The layering indicates that the IP would be called upon to provide services for the next higher protocol, TCP, and to call upon the network interface for

services required by IP. In the remote login process, the TCP segment(header and possibly data) is the data portion of the IP datagram. IP would then call on an Ethernet device driver to transmit the IP datagram. At the receiving host, there must be an IP protocol layer that properly interprets the header information for its next higher layer.

When a user wants to send data to a remote station on a long haul network, he provides the application layer with the name of the desired destination. It is the job of layers above IP to map or change that name into a network address. IP needs to know the "where" of the destination rather than the "who". IP depends upon lower level modules to provide the route for its datagram to follow across the network.

2. IP Specification

Figure 4.1 shows the IP header format. A discussion of the fields follows. The **version** field is four bits long and shows the version number used for the information contained in this header. The current version of IP header is 4. The **IHL** field, 4 bits, is the length of the internet header in 32 bit words and is used to determine where the IP data bytes begin in the datagram. The minimum length of the IP header is 5. In the remote login process, the IHL is always 5 (20 bytes of header). The **type of service** tells how the datagram is to be treated during its trip throughout the network (how important is this

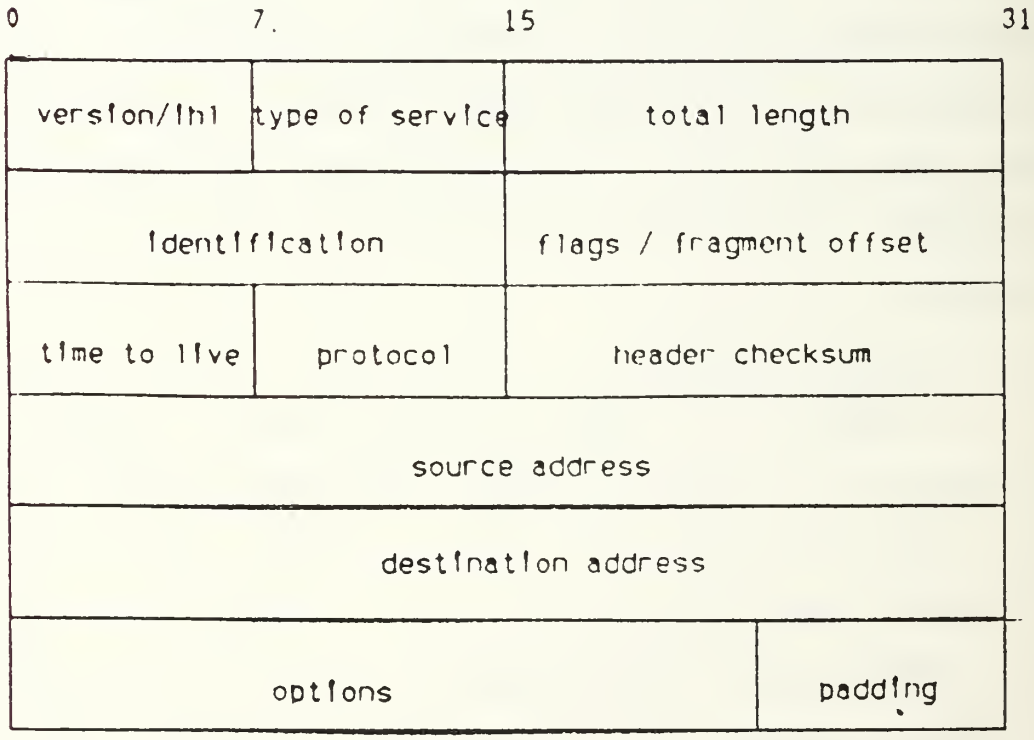

Figure 4.1 Internet Header Format

datagram). The remote login process uses only ØØ in this field which indicates normal delay, reliability, and throughput and routine precedence (as opposed to priority, immediate, flash, etc.). The **total length field** is the total byte count of the datagram, header and data. The two byte field allows a length of 65,535 bytes, but this is impractical for most hosts and networks. All hosts must be able to receive datagrams of at least 576 bytes.

Fragmentation is not really an issue when remotely logging in over an Ethernet to a host which is attached to the same Ethernet. The length of the data field in the Ethernet packet, up to 15ØØ bytes, is quite adequate to accomodate the protocols and data that are used in the remote login/terminal process. Fragmentation does become an issue when the remote login packet must cross long haul networks. The next three fields deal with fragmentation. The observed packets always had the **identification** bytes set to some host determined unique number associated with a particular source-destination pair and protocol for the time the datagram could be alive in the network. That value was incremented by Ø1 hexadecimal with each packet sent by a particular host. The **flags** field was always ØØ hexadecimal indicating each packet may be fragmented and that this particular packet was the last fragment (which matches with the identification field increasing by Ø1 hexadecimal with each packet). The **fragment offset** indicates where in the

original datagram this fragment belongs and thus was observed to always be ØØ hexadecimal in the remote login process.

The **time to live** field indicates the maximum time the datagram is allowed to remain in the Internet system. Its purpose is to cause undeliverable datagrams to be discarded and to bound the maximum datagram lifetime. As with fragmentation, this field is necessary if the packet must be sent over a long haul network. The observed value was always Ø7 hexadecimal. When remotely loging into a host on the same Ethernet, this field must contain a value of at least Ø1 hexadecimal because the IP specifies that packets with this field reading ØØ hexadecimal are to be destroyed.

The **protocol** field always reads Ø6 hexadecimal which indicates that the next higher protocol used in the data portion of the datagram is TCP. The **header checksum** is a checksum for the IP header only. It provides for high reliability in the communications. In the remote login process since TCP and its data form the data portion of IP, and this is all covered by the TCP checksum, the entire communication is covered by a checksum. The official wording of the checksum algorithm is "... the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero." [Ref. 3] See Appendix F of this thesis for further elaboration on the checksum.

The source address and destination address fields are grouped into three classes, class a, b, or c. The class that was obsserved in the remote login process was the class c address. The class is not a function of the remote login process; it is the address of the Naval Postrgraduate School on the ARPANET. The class c address has three bytes of network address and one byte for the local address. The network address of the VAX is c0 09 c8 03 hexadecimal, with the 03 indicating its local address. The Sun Workstation address is c0 09 c8 01 hexadecimal. The list of all valid hosts to which a particular UNIX implementation can talk can be found in the UNIX file "/etc/hosts"; the addresses appear in decimal form, not hexadecimal.

C. TRANSMISSION CONTROL PROTOCOL

1. Background

The Transmission Control Protcol (TCP) was designed to be used as a highly reliable host to host protocol in packet-switched networks and in interconnected systems of such networks. It provides for reliable inter-process communication between pairs of processes in host computers. The protocol makes few assumptions concerning the reliability of the protocols which belong to lower level layers. It is designed to operate above a wide spectrum of networks be they hardwired, packet switched, or circuit switched.

Figure 2.2 shows the position of TCP in the ARPANET network layers. TCP would be called upon by higher level protocols to provide service for them; and TCP would in turn call upon Internet for services. Whatever data or control bytes come to it from a higher layer would be added to the TCP header as data and passed to the IP layer. The TCP header and its data, if any, is called a segment, and it forms the data portion of an IP datagram in the remote login process. The destination host must also contain a protocol that recoginizes and evaluates the information contained in the TCP segment.

The TCP performs its services to higher protocols by providing mechanisms that ensure data transfer, reliability, flow control, multiplexing, and connections. TCP allows a continuous flow of data to/from each host. In general, a TCP would block and forward data at its own convenience, but there are certain times when a user must know that the data that has been sent to the TCP has been transmitted. This user determined transmission of data is called a "push". For instance, in the remote login/terminal process, the user forces or "pushes" the transmission of his UNIX command by the use of the carriage return key.

TCP can recover from data that is damaged, lost, duplicated, or delivered out of order. This is achieved by assigning a sequence number to each byte of data transmitted, and requiring positive acknowledgement of

reception from the receiving TCP. If not acknowledged, the segment is retransmitted after a certain time interval. These sequence numbers also allow the receiving host to order incoming packets so that they are provided to the next higher layer in the proper order. Damaged packets are handled by adding a checksum for the TCP segment; damaged packets are not processed. TCP is extremely robust. It provides recovery from any communications errors, providing the TCP is itself functioning.

TCP handles the problem of a fast sender swamping a slow receiver, flow control, by use of the window field. The window on an incoming segment indicates an allowed number of bytes that the receiver may transmit before receiving further permission. The window governs not how many packets are sent per unit of time, but how much data can be sent at any given time.

TCP allows for many processes within a single host to be communicating with the same remote host simultaneously by providing a set of addresses within each host called ports. This port address, when concatenated with a network and host address form what the ARPANET terms a socket. A pair of sockets, one in each host, uniquely identifies what is termed a connection. A connection is the process to process service provided by TCP. In the remote login process, each successful login to the remote host would form a separate connection from the user to the remote login port of the

VAX. Since the login process itself handles many users, many user could log in over the Ethernet to the remote login port. This can be demonstrated by using the Sun Workstation to log into the VAX system, then, using that connection, to remotley log back into the Sun, in turn using this new connection to remotely log into the VAX again, and so on. Since connections must be established between unreliable hosts over unreliable networks, a handshake mechanism with a clock-based sequence number intialization is used to prevent erroneous initialization of connections.

2. TCP Specification

TCP segments (header and data) are sent as part of an IP datagram; the TCP header immediately follows the IP header. Figure 4.2 shows the format for the TCP header. An explanation of the fields now follows.

The **source port and destination port** fieleds are used by TCP so that it may uniquely identify separate streams of data. This address, along with the address fields from the IP header form the address of the data stream that uniquely identifies it throughout the network. The port address for the UNIX login port is Ø2 Ø1 hexadecimal (513 decimal); a listing of this port address and all network services port addresses can be found in the UNIX file "/etc/services". UNIX evaluates all source ports of inbound packets to see if they are between Ø to Ø3 ff hexadecimal. Thus in the initial login request from the Sun Workstation, the source

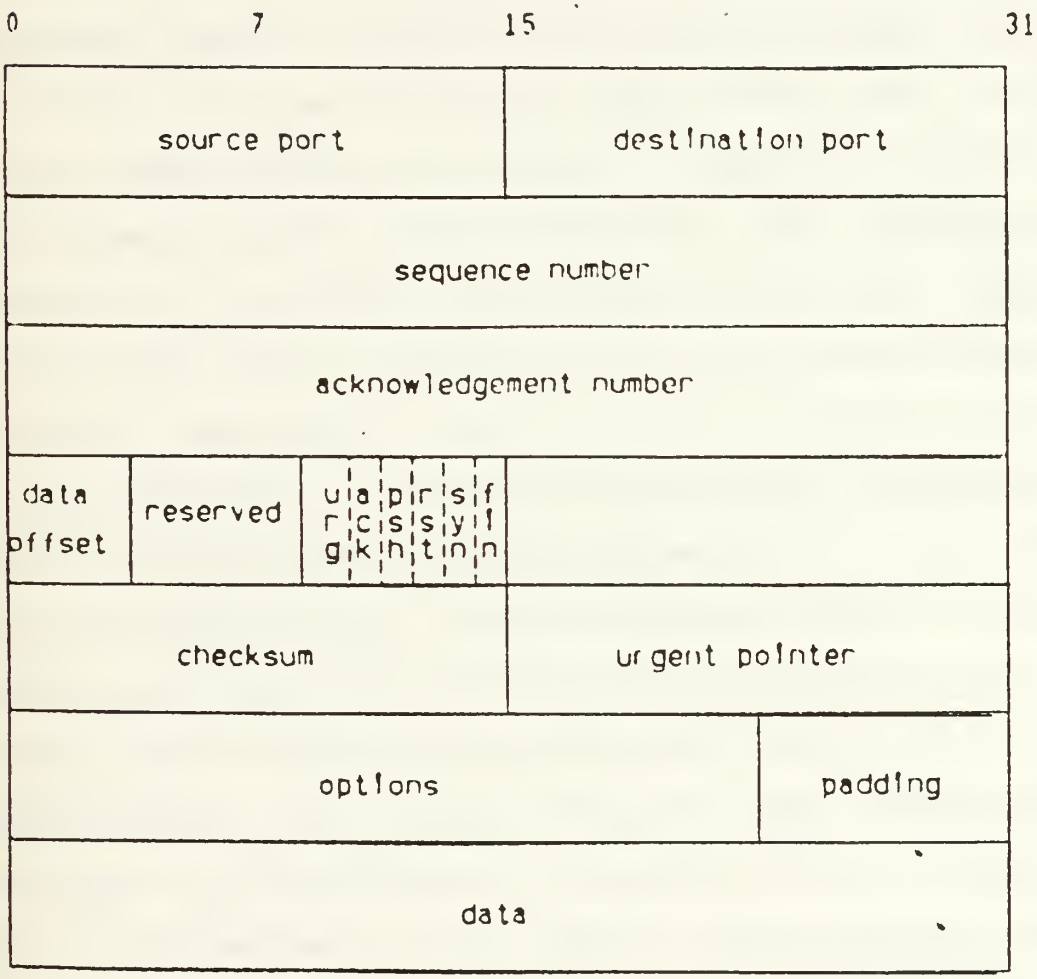

Figure 4.2 TCP Header Format

port is Ø3 ff hexadecimal and the destination port is Ø2 Ø1 hexadecimal. Should multiple users desire to log into the VAX from the Sun Workstation, each would use a destination port address of Ø2 Ø1 hexadecimal, but each would use a separate source address between Ø and Ø3 ff hexadecimal. This would uniquely identify each TCP connection and hence each separate login.

The **sequence** number field is a 32 bit number that refers to the first data byte in the data portion of that particular TCP segment. If there is no data present in the TCP segment, such as in the case of acknowledgement only packets, then the sequence number refers to the data byte that will next be sent by that socket. Since every byte of data is assigned a sequence number, the segment sequence number is the lowest unused number in the 32 bit sequence number space. The **acknowledgement** number field in a TCP segment is the sequence number of the next expected data byte of transmission in the reverse direction. Since every byte of data has a sequence number, each of them can be acknowledged. The simplest form of the acknowledgement procedure is to make it cumulative so that an acknowledgement number, call it X, indicates that all sequence numbers up to, but not including X, have been received. The initial sequence number is chosen based on a clock based algorithm. This enables TCP to detect duplicate packets from a previous instance of the connection. It is

important to note that a pure acknowledgement packet does not contain data and hence does not occupy sequence number space.

The **data offset** field is the number of 32 bit words in the TCP header. The **reserved** field is reserved for future use and is filled with Ø's. Each bit in the **control** field informs the TCP of a particular kind of event; each event requires different processing depending upon in which state the TCP is. More on TCP states will appear later in this thesis. The control bits indicate that the urgent or acknowledgement fields are significant, that data is to be pushed, to reset the connection, synchronize sequence numbers, and that no more data is to be sent over this particular connection.

The **window** field is the number of bytes of data that the sender of the segment is willing to accept from the other end of the connection. This is based on the size of the buffers available in a particular implementation and how much data is in each of the buffers at any given time. The observed values in the remote login were predominantly Ø8 ØØ hexadecimal, 2Ø48 decimal. The window is the mehanism by which a fast sender does not swamp a slow receiver. If a sender sends data that is more than can be received, that data would be discarded, thus adding to the network traffic by necessitating retransmission. By examining the sequence number and window size on an incoming packet, a TCP can

determine whether or not the number of bytes that it has to send can be received at the other end.

The **checksum** field uses the same algorithm for computation as the IP checksum; it covers a psuedo header, the TCP header, and the data portion of the segment. The pseudo header is a 12 byte header which contains the source and destination addresses from the IP header, a zero byte, the protocol number(Ø6, which indicates TCP), and the TCP length. The length is the length of the TCP header and data; it does not count the 12 bytes of pseudo header. Should the length be an odd number, then a zero byte is appended as padding in order to form the full 16 bit words for the checksum algorithm. Figure 4.3 shows the format of the pseudo header.

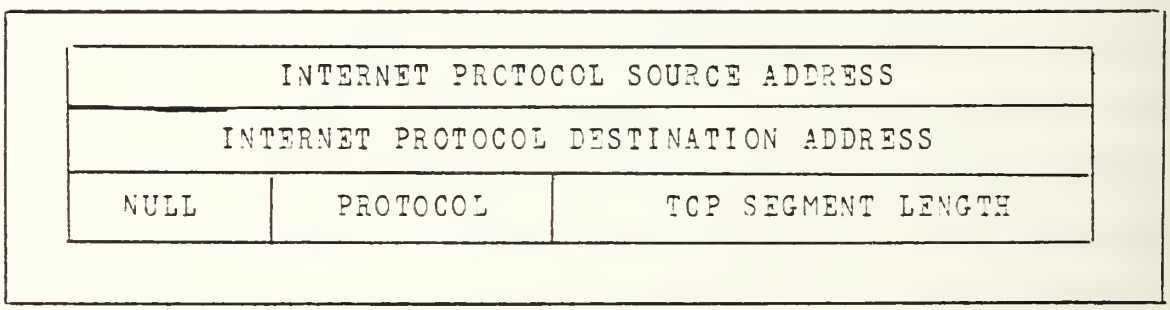

Figure 4.3 Pseudo Header Format

The **urgent pointer** field allows the sender to notify the receiver that some urgent data follows. The pointer is an offset value from the sequence number of the packet with the urgent control bit set; it points to the last byte of urgent data. In the remote login process, the **options** fields in

the TCP header are utilized only during the establishment of a TCP connection. The option observed specified the maximum segment size as being 1024 bytes. The padding field is used only to ensure that the TCP header contains full 32 bit words. The padding is the zero byte.

3. TCP States

A connection moves through a progression of states during its existence. Figure 4.4 is a finite state model of the TCP connection states. It should not be considered as a complete state diagram; it depicts only a few of the events that can cause a state change. The TCP may change states based on the events OPEN, CLOSE, SEND, RECEIVE, and ABORT and the control flags that are set in incoming segments. The listen state represents waiting for a connection request from any remote TCP and port. Syn-sent represents waiting for a matching connection request in response to the original connection request. The syn-received represents the waiting for an acknowledgement of the syn which was sent in response to an incoming syn. The established state represents an open connection over which data can now be sent in either direction. The fin-wait-1 state represents waiting for a termination request or the acknowledgement on one that was sent. The fin-wait-2 state represents only the waiting for a termination request. The close-wait state represents waiting for a termination request from the local user. The closing state represents the waiting for a

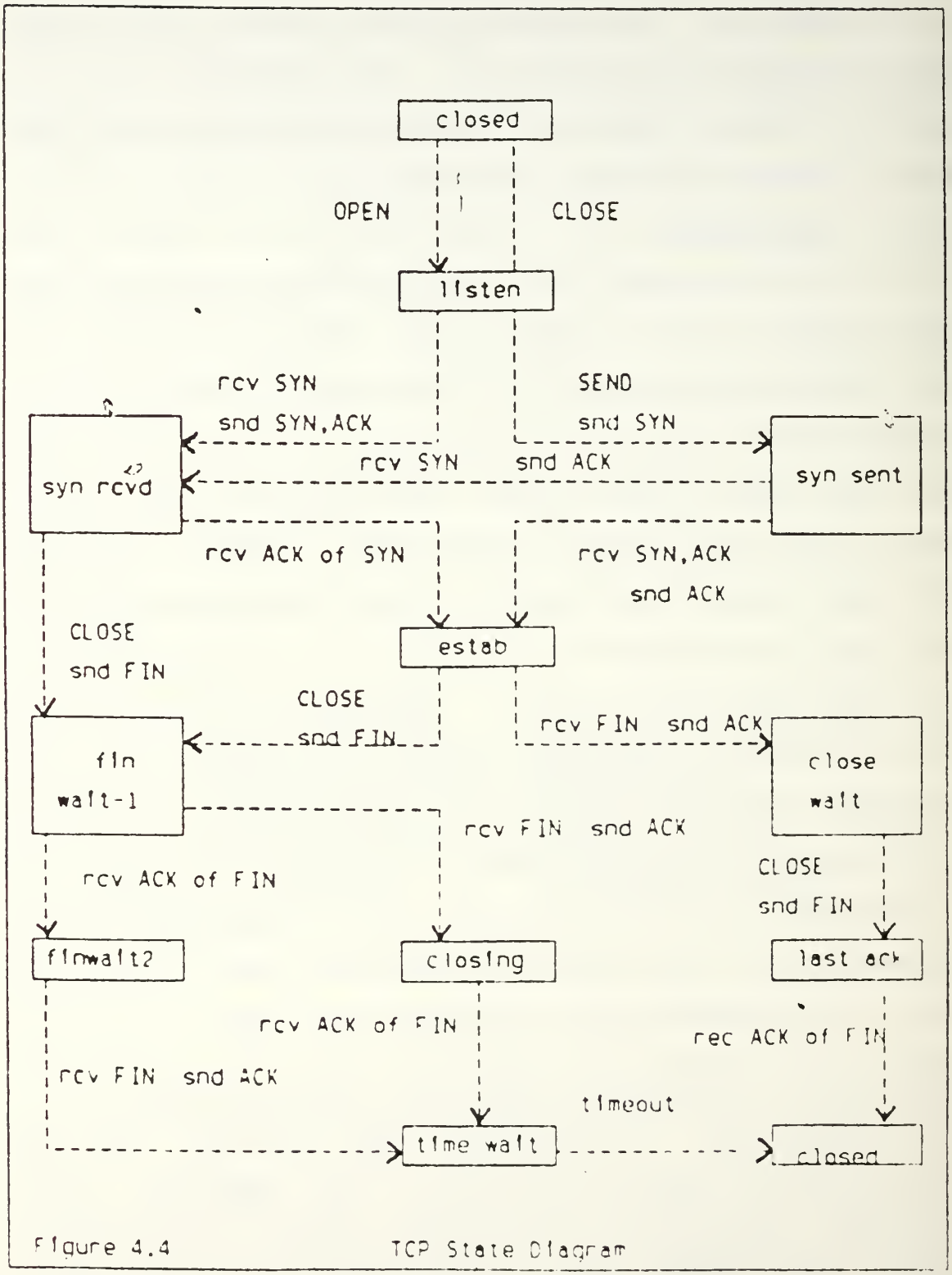

Figure 4.4 TCP State Diagram

connection termination request acknowledgement for the remote TCP. The last-ack state represens the wait for an acknowledgement of the connection termination request previously sent to the remote TCP. The time-wait state represents waiting for enough time to pass to make sure that the remote TCP has received acknowledgement of its connection termination request.

When a particular event takes place, the TCP response to that event depends upon the state of the TCP; the same event can cause different TCP reactions. The response to inbound packets is very complex; the algorithms covering the proper responses are found in Reference 3. The states of the TCP could be lumped together into three primary states; the first state has to do with getting a connection set up from one host/port to another, the second concerns the handling of inbound/outbound packets once established, and the third has to do with terminating the connection. The UNIX command NETSTAT will show, among other things, the state of each TCP connection.

4. Connection Establishment

For a connection to move to the established state, the two TCP's must synchronize on each other's initial sequence numbers. Since the initial sequence number from each socket is unique (they are clock generated, but not from a network wide clock), this synchronization is necessary so that the acknowledgement mechanism can function

properly. This synchronization is signaled by the syn's control bit in the TCP header being set. The synchronization requires each TCP to send its own uniquely determined sequence number and to receive acknowledgement that it has been received by the other end of the proposed connection. An example of the connection establishment is shown in Figure 4.5. The initial TCP from B can contain both the acknowledgement and the initial sequence number thus resulting in a total of three packets. This is where the exchange get its name of the "three way handshake".

The demonstration program in this thesis does follow the above synchronization of sequence numbers and produces a connection that moves into the "established" state. It does

a.	A ---> B	syn, my sequence number is X
b.	A <--- B	ack, your sequence is X
		syn, my sequence number is Y
c.	A ---> B	ack, your sequence number is Y

Figure 4.5 TCP Connection Establishment

not handle, in a TCP sense, the production of transmission packets nor the processing of inbound packets in the established state nor for connection termination. The algorithms for all event processing are contained in Reference 3.

D. UNIX PROTOCOLS

1. Background

All of the protocols discussed thus far have been very general and can be applied to many networking situations. The Ethernet, IP, and TCP protocols do not pertain directly to the remote login/terminal process per se. They are just as valid for sending mail, transferring files, etc. as they are for the remote login/terminal. The UNIX protocols that will now be discussed are specific to the remote login/terminal process.

2. UNIX Remote Login Protocol

A description of the remote login protocols can be found in the System Maintenance section of Reference 4. The servers for the protocols are RLOGIND(8C) and RSHD(8C). When the UNIX receives a request for a remote login over a network, it is RLOGIND that is listening at the rlogin port for such requests (listen, in the sense of a TCP state diagram). Upon receiving a request, RLOGIND along with RSHD checks to see if the requestor's source port is within the range 1-1023. They read characters from the socket up to the null byte (00 hexadecimal) and interpret them as ASCII characters. They check to ensure that the requestor's address is one contained in their host's name data base. A null terminated user name of at most 16 characters is next read and is considered to be the user name for the host machine; a second null terminated user name is read and is

considered the user name on the requestor's machine. Next, a null terminated command to the UNIX command shell is retrieved. The user name is then checked against the list of authorized users. When all the above steps are successful, the null byte is returned over the connection to the requestor. The connection is now passed on to the regular login process for the verification of the password.

The above steps are those described in the UNIX operating system manuals. Observations of the conversation over the Ethernet during the remote login process reveal the following. A three way handshake, according to TCP, etablishes a connection from the Sun Workstation to the remote login port of the VAX system. (A UNIX NETSTAT command issued after only the three way handshake packets have been transmitted will show that the login port of the VAX has a network connection to the foreign address, Sun1. The state of the connection will read 'established'.) In the next packet, the Sun Workstation then pushes the null byte; then a new packet which has the user name as data, then the next packet which also has the user name as data, then the terminal type/baud rate appears as data to the VAX. The Vax will then send "Password" to the Sun Workstation as data to be displayed on the Sun terminal. This is the first displayable data that is passed to the Sun Workstation. The Sun will now be using the protocol discussed below.

3. UNIX Terminal Protocol

Terminals connected to a VAX 11/780 running 4.2 BSD UNIX are full duplex, character at a time connections that normally terminate at a communications interface processor. When typing at the terminal, each key stroke is sent separately to the UNIX; it is returned or "echoed" to the terminal so that it can be displayed on the screen. The echo is displayed, not the key when struck. It is this echo feature that enables the user to type in a password without it appearing on the screen. The operating system knows when the user is sending a password, so it does not return or echo the keystroke to the terminal. When using the remote terminal feature of the UNIX, the remote terminal is treated in the same manner as described above. The remote terminal process transmits each keystroke as a single packet; the provider of the service replies with a single packet that echoes the keystroke (this echo is then acknowledged in an ack only segment). When the VAX system is sending lines of text to be displayed on the remote terminal, it will send at most one line of data in each packet with ASCII carriage return and line feed characters as the line terminators.

4. Trailer Protocols

Section 7Ø of Reference 4 contains information on the use of trailing protocols. They appeared after the data bytes of the TCP when the length of IP, TCP, and data was less than the minimum Ethernet data length of 46. For

instance, if the remote login/terminal ARPANET packet was only 40 bytes long, there was a 6 byte trailer added to the packet to make it a total of 46 bytes long. No investigation into the contents of these protocols was conducted.

5. Summary of Remote Login/Terminal

Figure 4.6 depicts the protocols that have been investigated during the course of this thesis. There would be other layers above these, but they were neither investigated nor programmed in the demonstration program.

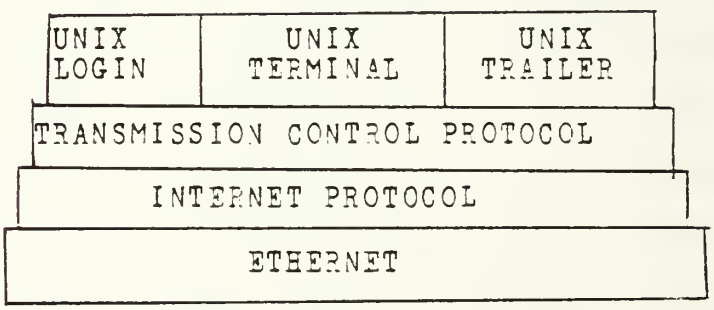

Figure 4.6 UNIX Remote Login Protocols

V. TEST OF VIABILITY

A. DEMONSTRATION PROGRAM

1. Goals of the Demonstration Program

In order to test the viability of a remote login to the UNIX over an Ethernet from a CPM based process, a demonstration program was devised that mimicked the remote login conversations between the Sun Workstation and the VAX UNIX system. The demonstration program used the Sun Workstation's Ethernet, ARPANET, and login port addresses. The program did not attempt a full implementation of the IP/TCP protocols nor of the UNIX remote login/terminal protocols. The goal of the program was to:

a. establish a TCP connection between the CPM based process and the login port of the VAX UNIX machine (three way handshake) and

b. to proceed through the UNIX remote login protocols far enough that the VAX UNIX transmitted the request "Password" (requestor has successfully passed through the remote login and has been handed off to the standard UNIX login process).

The successful accomplishment of this could be determined through the use of the UNIX command NETSTAT and the examination of the packets that were sent by the VAX system in reply to the demonstration program. If the Sun Workstation is turned off and then the demonstration program is run, executing the command NETSTAT will show that the VAX UNIX has a TCP connection "established" to its login port from the remote host Sun1. Examination of the packets

returned to the demonstration program by the VAX UNIX system shows acknowledgement by the VAX system of all data received and the word "Password" passed as the data portion of its packets. The demonstration program went no further; the user's password was not sent to the VAX, no remote terminal protocols nor TCP connection termination protocols were attempted. Since full protocols were not programmed, the program assumes that a normal, trouble free remote login/terminal sequence will occur. Thus there can not be any other packets on the Ethernet not concerned with the remote login process.

Numerous observations of the remote login were obtained by the program LISTEN. The three way handshake was observed as described in the previous chapter. The demonstration program sent out a request for a connection to the VAX UNIX login port, waited for the return "syn,ack" from the VAX, then sent an acknowledgement of that packet. When forming this acknowledgement packet, the demonstration program, DEMO, only processed the inbound packet for its sequence number.

Figure 5.1 illustrates the observed sequence for the UNIX remote login protocol. DEMO sent out its packets assuming this would be the sequence of events; it pushed the four packets, waited for the ack and then null byte from the VAX, and then processed the VAX packet for its sequence number with which it could form an acknowledgement packet.

2. Demonstration Program Results

The demonstration program was successful in meeting its goals. A TCP connection was established and the UNIX

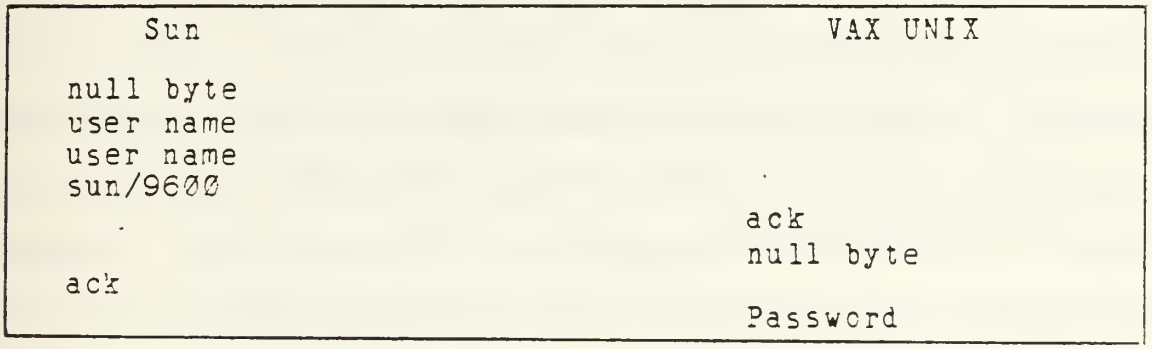

Figure 5.1 UNIX Remote Login Sequence

remote login protocols were successfully negotiated and the request passed onto the normal login process of the VAX UNIX; the requestor was asked for his password.

3. Significant Program Development Anomalies

During the development of both the demonstration program and the LISTEN program, several important anomalies developed; one concerned the network protocols, another the the NI3010 board, and lastly the asynchronous interrupts while CPM is executing i/o.

When DEMO is being used, it initiates a TCP connection with the VAX UNIX system in accordance with the specifications. However, it does not terminate the connection according to the TCP protocol. When DEMO is finished, the connection is one sided; the VAX still thinks that it has a connection to the Sun. The connection eventually times out when the VAX fails to receive any more

packets on that connection. The next time the Sun attempts a remote login, the VAX system replies to the initial request not with an "ack,syn' packet but with a broadcast packet. The protocol for this was not investigated, but a cursory examination revealed that the intent of the packet appeared to be an update to VAX UNIX system address tables that were altered when the previous connection from the same source terminated abnormally. For further program development, this can be avoided by merely logging back into the VAX from the Sun before attempting the next running of DEMO.

When running either LISTEN or DEMO, the program can run to normal termination or be terminated early if hung up or the desired number of packets has not been received. On the AEGIS multiuser system, this can be accomplished by either resetting the MULTIBUS cluster or resetting only the 86/12 SBC in use. If either program has been terminated early by resetting the 86/12 SBC, then the next time either one is run an extra byte of data is transferred from the board into the extended TPA. When the receive data block is overlaid on this portion of memory, it is off by this one byte, thus causing the program to improperly determine the size of the received packet. This can be avoided by always resetting the MULTIBUS cluster before running the program. This has the same effect as turning off the power to the board and then turning it back on. This power down

operation will wipe out all buffers and registers on the NI3010 board. Both LISTEN and DEMO will issue a warning message when there is a size error and then terminate.

PL/I-86 input/output routines are not re-entrant. Therefore, when doing any i/o, it is necessary to do so in the NI3010 board interrupt handler or make calls through this handler. Since the first step of the board interrupt handler is to disable the board interrupts, all code will execute to completion without interruption; therefore, no i/o routines will be interrupted. If they were, unpredictable results on the terminal will occur.

VI. CONCLUSIONS

The goals of the thesis were met by the demonstration program's showing the viability of a microcomputer based process logging into a VAX UNIX system over the Ethernet. The basic information on the protocols used is available; they can be implemented on a 16 bit microprocessor; PL/I is sufficiently flexible to allow the linking of the assembly language routines necessary to write the device driver for the NI3010 board and to perform 32 bit arithmetic and 16 bit checksums.

Now that the fundamental concepts have been recorded, it is necessary to implement, in a modular, layered approach the full remote login/terminal capabilities. Once accomplished, it would be possible to use the program for communication over the ARPANET. By logging into the Vax UNIX system, its full networking capabilities can be utilized.

APPENDIX A

PROGRAMMING NOTES

There are several features of PL/I-86 and LINK-86 that are important to understand so that the programs and linking insructions for them can be understood.

The linking command for both LISTEN and DEMO are quite similar; the only real differences are the assembly language modules that are linked into each. The main point to understand about them is the parameter listing for the PL/I-86 program. The linking commands used were: "link86 listen[i]" and "link86 demo[i]" where the "[i]" option was an input file for the linker. The input file was:

```
demo [data[abs[800],m[0],ad[82],map[all]],
cksum, add32bit,xternasy
```

The first parameter for DEMO says to load the data segment register of the 8086 microprocessor with 800 hexadecimal. This forces the data segment of 64K bytes to run from the base address of 8000 hexadecimal to 17fff (remember that address registers are left shifted by four bits before the offset is added). Thus one half the data segment is on 86/12 SBC onboard memory and the other half of the data segment is located in the extended TPA. The reason for this straddle of onboard/extended memory is that the onboard memory is strapped so that it is unaccessible from MULTIBUS. Since the NI3010 needs to transfer packets to/from host

memory, it must do so to the portion of the data segment which is in the extended memory.

The "m[0]" data parameter tells the linker to allocate no additional memory for the data section of the command file. This prevents the overwriting of command file data by the transfer of packets into the extended TPA which lies within the data segment address space.

The "ad[82]" data parameter gives the total number of additional paragraphs to allocate to the data segment of the command file. This enables the user to run either LISTEN or DEMO within the local memory of the iSBC 86/12.

The combination of the linker's loading of the data segment register with 800 hexadecimal, the PL/I-86 "unspec" built in function, and the PL/I-86 pointer based structures, allows the programmer to locate a template in the extended TPA for the transmit data blocks and receive data blocks for the NI3010 DMA operations. For instance, in the LISTEN program, the linker has loaded the data segment register with 800 hexadecimal, and the receive data block is based on a pointer whose value is forced to be 8000 hexadecimal by the PL/I-86 statement "unspec(x) = '8000'b4;" Thus the address of the receive data block is the sum of the 8000 hexadecimal and the 800 hexadecimal (left shifted); this yields an address of 10000 hexadecimal, which is the first byte of the extended TPA.

An important point to remember in network implementations is that only binary information is received or transmitted. The interpretation of each byte of binary Ø's and 1's is up to the implementation. A byte could be considered to be a character, an integer, or control information. For this reason, all declarations of structures which are used as overlays for the networking packets are declared as "bit(8)" in PL/I-86. The interpretation of these as characters or control is no real problem. However, when a particular byte, or more normally a group of two bytes, are to be considered as integers, then it was necessary to overlay the bit string with a based variable whose type was declared as "fixed", "fixed bin(7)", or "fixed bin(15)" in PL/-86. This is because run time conversions of bit to fixed values caused unexpected results.

Another important point is the manner in which PL/I-86 stores integers and hexadecimal values. A two byte integer is stored with the least significant byte preceding the most significant byte in memory. Thus overlaying a "fixed" or "fixed bin(15)" over the length bytes of the NI3010 receive frame is acceptable because the protocol says that the least significant byte is first in memory followed by the most significant byte, just as in PL/I-86. This will not work for the length bytes as they are stored for the Internet Protocol. There the two byte integer value is

transferred into/out of memory with the most significant byte first followed by the least significant byte. Therefore a two byte PL/I-86 variable cannot be overlayed on the IP length bytes. Similarly, the hexadecimal value 12 34 is stored in memory as 34 12. This is important when storing 16 bit hexadecimal values in bit(16) variables such as the TCP source and destination port addresses.

APPENDIX B

NI3010 DEVICE DRIVER ASSEMBLY LANGUAGE MODULES

date:20 November 1984

author: David J Brewer

These 8086 Assembly language modules were developed by Brewer for his thesis.[Ref. 5] They are linked into any program that needs to utilize an Interlan NI3010 Ethernet Controller Board. They read/write to the various ports of the NI3010, initialize the interrupt 5 for the CPU, and enable/disable the CPU interrupts.

```
;                          XTERNASY


extrn hl_interrupt_handler : far

public write_io_port
public read_io_port
public write_bar
public initialize_cpu_interrupts
public enable_cpu_interrupts
public disable_cpu_interrupts
;*****************************************************************

write_io_port:

    ; Parameter Passing Specification:

    ;                       entry                       exit
    ;
    ; parameter 1      <port address>              <unchanged>
    ;
    ; parameter 2      <value to be outputted>     <unchanged>
    ;

        dseg

        port_address    rb   1

        cseg
```

```
            push bx! push si! push dx! push ax
            mov  si, [bx]
            mov  al, [si]
            mov  port_address, al
            mov  si, 2[bx]
            mov  al, [si]
            mov  dl, port_address
            mov  dh, 00h
            out  dx, al
            pop ax! pop dx! pop si! pop bx
            ret

;*************************************************************************

read_io_port:

     ; Parameter Passing Specification
     ;
     ;                      entry                    exit
     ;
     ; parameter 1    <port address>          <unchanged>
     ; parameter 2    <meaningless>        <register value>

            cseg
            push bx! push si! push dx! push ax
            mov  si,  [bx]
            mov  al,  [si]
            mov  port_address,  al
            mov  si, 2[bx]
            mov  dl, port_address
            mov  dh,  00h
            in   al,  dx
            mov  [si], al
            pop  ax! pop dx! pop si! pop bx!
            ret

;*************************************************************************

write_bar:

    ; Parameter Passing Specification
    ;
    ; parameter 1 :the address of the data block to be
    ;              transmitted or received.

            dseg
```

```
e_bar_port      equ  0b9h
h_bar_port      equ  0bah
l_bar_port      equ  0bbh
temp_e_byte     rb   1
temp_es         rw   1

cseg

; This module computes a 24 bit address from a 32 bit
; address - actually it's a combination of the ES register
; and the IP passed via a parameter list.

    push bx! push ax! push cx! push es! push dx! push si


        mov  dx, 0800h        ; common memory segment
        mov  es, dx
        mov  temp_es, es
        mov  dx, es
        mov  si, [bx]
        mov  ax, [si]
        mov  cl, 12
        shr  dx, cl
        mov  temp_e_byte,  dl
        mov  dx, temp_es
        mov  cl, 4
        shl  dx, cl
        add  ax, dx
        jnc  no_add
add_1:  inc  temp_e_byte
no_add: out  l_bar_port, al
        mov  al, ah
        out  h_bar_port, al
        mov  al, temp_e_byte
        out  e_bar_port, al
        pop si! pop dx! pop es! pop cx! pop ax! pop bx
        ret


;-----------------------------------------------------

initialize_cpu_interrupts:


    ; Module Interface Specification:

    ;     Caller:        Ethertest(PL/I) Procedure

    ;     Parameters:    NONE
```

```
    initmodule cseg common
               org 114h
               int5_offset   rw 1
               int5_segment  rw 1

               cseg
               push bx
               push ax
               mov  bx,  offset interrupt_handler
               mov  ax,  0
               push ds
               mov  ds,  ax
               mov  ds:int5_offset, bx
               mov  bx, cs
               mov  ds:int5_segment, bx
               pop  ds
               pop  ax
               pop  bx
               sti
               ret

;-----------------------------------------------------

enable_cpu_interrupts:

     ; Module Interface Specification:

     ;     Caller:         Ethertest(PL/I) Procedure

     ;     Parameters:     NONE

               sti
               ret

;-----------------------------------------------------

disable_cpu_interrupts:

     ; Module Interface Specification:

     ;     Caller:         Ethertest(PL/I) Procedure
     ;     Parameters:     none

               cli
               ret
```

```
;-------------------------------------------------------

interrupt_handler:

                ; IP, CS, and flags are already on stack
                ; save all other registers

                push ax
                push bx
                push cx
                push dx
                push si
                push di
                push bp
                push ds
                push es

                call hl_interrupt_handler
                     ; high level source routine
                     ; In Ethertest Module (PL/I)


                ; restore registers

                pop es
                pop ds
                pop bp
                pop di
                pop si
                pop dx
                pop cx
                pop bx
                pop ax
                sti
                iret


end
```

APPENDIX C

LISTING OF LISTEN PROGRAM

```
listen:                    procedure options (main);

/*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

                        date: 20 november 1984

                          John Donald Reeke

This  program  will  capture all packets  that  are  on  the
Ethernet.   It does this by initializing the NI3010 board to
the promiscuous mode.   The packets are placed one after the
other in memory starting at 8000 hex,  the first byte of the
extnded tpa.   The contents of the packets are output to the
file  PACKET.DAT but not to the terminal.   The user selects
either  hexadecimal or binary output and also the number  of
packets  the  program will receive  before  termination.  An
anomally  exits in the NI3010 board.  Always hard reset  the
MULTIBUS  backplane  before using this  program.   When  the
anomally is present,  an extra byte,  00 hex,  precedes  the
first packet dma'ed from the board.  This causes the receive
block  overlay to be off by this one  byte;  therefore,  the
size  of the first packet is not properly determined by  the
program.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/


   declare

       1 receive_data_block based(rec_blk_ptr),
            2 frame_status              bit (8),
            2 null_byte                 bit (8) ,
            2 frame_length_lsb          bit (8) ,
            2 frame_length_msb          bit (8) ,
            2 destination_address_a     bit (8) ,
            2 destination_address_b     bit (8) ,
            2 destination_address_c     bit (8) ,
            2 destination_address_d     bit (8) ,
            2 destination_address_e     bit (8) ,
            2 destination_address_f     bit (8) ,
            2 source_address_a          bit (8) ,
            2 source_address_b          bit (8) ,
            2 source_address_c          bit (8) ,
            2 source_address_d          bit (8) ,
```

```
          2 source_address_e          bit (8) ,
          2 source_address_f          bit (8) ,
          2 type_field_a              bit (8) ,
          2 type_field_b              bit (8) ,
          2 data (1500)               bit(8),
          2 crc_msb                   bit (8) ,
          2 crc_upper_middle_byte     bit (8) ,
          2 crc_lower_middle_byte     bit (8) ,
          2 crc_lsb                   bit (8) ,

      size_ptr  pointer,
      size  fixed bin(15) based (size_ptr),
      fill_xtpa(15000) char based(rec_blk_ptr),
      output_type char(1),
      user_desires fixed,
      packet file,
      rec_blk_ptr pointer,
      number_of_packets fixed static init(0),
      copy_ie_register bit(8),
      i  fixed bin (15),
      reg_value bit (8) ,
      border (80) char (1) static initial ((80)'-'),


/*   Modules external to this module */

      write_io_port entry (bit (8) ,bit (8) ),
      read_io_port  entry (bit (8) ,bit (8) ),
      initialize_cpu_interrupts     entry,
      enable_cpu_interrupts         entry,
      disable_cpu_interrupts        entry,
      write_bar entry (pointer);

/*   end module listing  */


      %replace

          next_page                 by '^l',
          clearscreen               by '^z';

          /*  codes specific to the Intel 8259a
              Programmable Interrupt Controller (PIC)   */

                      icw1_port_address         by 'c0'b4,
      /* note that */  icw2_port_address         by 'c2'b4,
      /* icw2,icw4,*/  icw4_port_address         by 'c2'b4,
      /* and ocw   */  ocw_port_address          by 'c2'b4,
      /* use same  */
      /* port addr */
```

```
                    /* note: icw ==> initialization
                                     control
                                     word

                           ocw ==> operational
                                     command
                                     word              */


                    cw1                        by '13'b4,

                     /* single PIC configuration, edge
                        triggered input              */

                    cw2                        by '40'b4,

                     /* most significant bits of vectoring
                        byte; for an interrupt 5,
                        the effective address will be
                        (icw2 + interrupt #) * 4 which
                        will be (40 hex + 5) * 4 =
                                 114 hex             */

                    icw4                       by '0f'b4,

                     /* automatic end of interrupt
                        and buffered mode/master   */
                    ocw1                       by '9f'b4,

                     /* unmask interrupt 5 (bit 5) and
                        mask all others              */

                  /* end 8259a codes */


/* include constants specific to the NI3010
   board                                          */

%include 'listen.dcl';  /* see APPENDIX G */
```

```
/*****************************************************************/

                        /*  Main Body */


  /*
      disable the ni3010 board so that it does not interrupt
      this new initialization
  */
call write_io_port(interrupt_enable_register,
            disable_ni3010_interrupts);


  /*
     initialize ni3010, PIC, and CPU interrupts
  */
call read_io_port (command_status_register,reg_value);
call perform_command(go_offline);
call perform_command(reset);
call initialize_pic;
call initialize_cpu_interrupts;

  /*
     set the receive block base address at 8000 hex,
     the first memory location in the extended tpa;
     fill the extended tpa with '*'
  */
unspec(rec_blk_ptr) = '8000'b4;
do i = 1 to 15000;
      fill_xtpa(i) = '*';
end;


  /*
     user interface
  */
put list (clearscreen);
put skip;
put edit ((border (i) do i = 1 to 80)) (a);
put skip list('ENTER 1 FOR HEX PRINTOUT:ENTER 2 FOR BINARY');
put skip;
get list(output_type);
put skip;
open file (packet) stream output;
put skip list('How many packets do you wish to receive');
put skip;
get list(user_desires);
```

```
  /*
     put ni3010 online and prepare to receive packets
  */
call perform_command(go_online);
call perform_command(set_promiscuous_mode);
copy_ie_register = receive_block_available;
call write_io_port (interrupt_enable_register,
                            receive_block_available);


  /*
     wait  until user specified number of have been  packets
     received
  */
do while (number_of_packets < user_desires );
end;


  /*
    reset the ni3010 to clear buffer; close file
  */
call perform_command (reset);
put skip (3);
put edit ((border (i) do i = 1 to 80)) (a);
put skip (2);
close file(packet);

                  /* end main body */


/*****************************************************************
This module initailizes the programmable interrupt
controller on the 86/12 SBC
*****************************************************************/

initialize_pic:   procedure;

  declare

     write_io_port entry (bit (8) , bit(8));
     call write_io_port (icw1_port_address,icw1);
     call write_io_port (icw2_port_address,icw2);
     call write_io_port (icw4_port_address,icw4);
     call write_io_port (ocw_port_address,ocw1);

end initialize_pic;
```

```
/*************************************************************
This  module sends to the NI3010 board the command  received
in the parameter list.   It then waits to see if the command
has been successfully carried out by the board.
*************************************************************/

perform_command:        procedure (command);

  declare
        command bit (8) ,
        reg_value bit (8) ,
        srf bit (8) ,
        write_io_port entry (bit (8) ,bit (8) ),
        read_io_port  entry (bit (8) ,bit (8) );


     srf = '0'b4;
     call write_io_port (command_register,command);
     do while ((srf & '01'b4) = '00'b4);
        call read_io_port (interrupt_status_reg,
                           srf);
     end;  /* do while */
     call read_io_port (command_status_register,
                   reg_value);
     if (reg_value > '01'b4) then
     do;
        /* not (SUCCESS or SUCCESS with Retries) */

       put skip edit ('*** ETHERNET Board Failure ***')
                     (col(35),a);
       stop;
     end;

end perform_command;
```

```
/*************************************************************
This module is called by the assembly language interrupt
routine.  It handles three NI3010 interrupts: the receive
block available, receive dma done, and the transmit dma done
interrupts.  It also checks for the NI3010 anomaly and
prints out an error message and terminates the program
should the anomaly exist.  The receive block is
repositioned in memory after each packet has been dma'ed to
memory.
*************************************************************/


 HL_interrupt_handler:    procedure external;

  declare

     write_io_port entry (bit (8) ,bit (8) ),
     read_io_port entry (bit (8) ,bit (8) ),
     enable_cpu_interrupts          entry,
     disable_cpu_interrupts         entry,
     write_bar entry (pointer);



        /*
           don't let anything interrupt
        */
     call disable_cpu_interrupts;
     call write_io_port(interrupt_enable_register,
                        disable_ni3010_interrupts);


     if (copy_ie_register = receive_block_available) then
        do;

        call write_bar (addr(receive_data_block));
        call write_io_port(high_byte_count_reg,'06'b4);
        call write_io_port(low_byte_count_reg,'00'b4);

           /* initiate receive DMA */

        copy_ie_register = receive_dma_done;
        call write_io_port(interrupt_enable_register,
                           receive_dma_done);

        end;   /* if receive block available  */
```

```
      else
         if (copy_ie_register = receive_dma_done) then
            do;
               number_of_packets=number_of_packets + 1;
               size_ptr=
                  addr(receive_data_block.frame_length_lsb);
                  /*  check  for  ni3010  anomally  and,   if
                      present  set number of packets so  that
                      we  drop  through  to the  end  of  the
                      program  */
               if (size > 1522 | size < 64) then
                  do;
                     put skip list('SIZE ERROR');
                     number_of_packets = user_desires;
                   end;
               else
                  call print_packet;
                  /* below repositions the receive data
                     block overlay  */
               rec_blk_ptr =
                  addr(receive_data_block.data(size - 4));
         If (number_of_packets >= user_desires) then
            call write_io_port(interrupt_enable_register,
                               disable_ni3010_interrupts);
         else
            do;
               copy_ie_register = receive_block_available;
               call write_io_port(interrupt_enable_register,
                                  receive_block_available);
            end;

      end;  /* if receive dma done  */

      else
         if (copy_ie_register = transmit_dma_done)
            then do;
               copy_ie_register = receive_block_available;
               call write_io_port(interrupt_enable_register,
                                  receive_block_available);

            end;   /* if transmit dma done  */
end HL_interrupt_handler;
```

```
/*************************************************************
This  module is called from the HL  interrupt  routine.   It
prints  each packet received into the file PACKET.DAT.   Its
output  is either in hexadecimal or binary,  as selected  by
the user.
*************************************************************/

print_packet:           procedure;


     /*
        print out destination,source,frame status, frame
        length, and frame type.
     */
  if (receive_data_block.destination_address_f = '27'b4)
     then put file(packet) skip list
          ('                    from VAX to SUN');
  else if (receive_data_block.destination_address_f =
                                              '7f'b4)
     then put file(packet) skip list
          ('                    from SUN to VAX');
  else    put file(packet) skip list('BROADCAST PACKET');

  put file(packet) skip;
  put file(packet) edit('frame status =        ',
        receive_data_block.frame_status)(skip,a,b(8));
  put file(packet) skip list
        ('frame length =        ',size);

  put file(packet) edit('type_field_a =        ',
     receive_data_block.type_field_a)(skip,a,b(8));
  put file(packet) edit('type_field_b =        ',
     receive_data_block.type_field_b)(skip,a,b(8));


     /*
        this section prints out the IP and TCP headers
        in either hex or binary
     */
  put file(packet) edit('the data is:')(skip,a);
  put file(packet) skip;
  do i  = 1 to 20;
     put file(packet) skip;

        /* is user selected hexadecimal output */
     if (output_type = '1') then
        put file(packet) edit('data ',i,'= ',
        receive_data_block.data(i),'data ',(i + 20),'= ',
        receive_data_block.data(i + 20))
        (a,f(4),a,b4(2),col(30),a,f(4),a,b4(2));
```

```
      else
            put file(packet) edit('data ',i,'= ',
            receive_data_block.data(i),'data ',(i + 20),'= ',
            receive_data_block.data(i + 20))
            (a,f(4),a,b(8),col(30),a,f(4),a,b(8));
   end;
   put file(packet) skip;


      /*
         prints out data portion of TCP in either hex
         or binary
      */
   do i  = 41 to (size - 18);
      put file(packet) skip;

         /* if user has selected hexadecimal output */
      if (output_type = '1') then
         put file(packet) edit('data ',i,'= ',
            receive_data_block.data(i))(a,f(4),a,b4(2));
      else
         put file(packet) edit('data ',i,'= ',
           receive_data_block.data(i))(a,f(4),a,b(8));
   end;

   put file(packet) skip list (next_page);


end;   /* print_packet */



end; /* procedure listen */
```

APPENDIX D

LISTING OF DEMONSTRATION PROGRAM

```
demo:   procedure options (main);

/*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

                    date: 20 November 1984

                  author: John Donald Reeke

This  program demonstrates that a CPM-86 process running  on
the  AEGIS  multiuser  system  can remotely  login  over  an
Ethernet  to  the VAX 11/780 running 4.2  BSD  UNIX.  It  is
designed to mimic the remote login of the Sun Workstation to
the  VAX  and  therefore uses the  Sun's  Ethernet  address,
source address, and source port.  The program is intended to
only  go as far in the remote login process as is  necessary
for  the  VAX to send the data "Password" to  this  process.
Inbound  packets from the VAX are processed only  for  their
sequence number and whether or not they contain data.
  It initializes the ni3010 board and then asks the user how
many  packets to capture before termination of the  program.
Since  this  program captures all packets on  the  Ethernet,
including  its  own transmissions,  a minimum of 22  packets
should  be  selected.  Next the user is  prompted  for  the
destination  of the remote login.  All packets received are
placed consecutively in memory; no contents are displayed to
the  terminal.  It  is necessary to look in memory  with  a
debugging tool such as DDT-86 to examine the packets.
  Before executing this program, ensure that the VAX and the
Sun  have  logged in to each other  to  preclude  broadcast
packets,  as discussed in the body of the thesis, from being
utilized  in the login procedure.  Also,  the Sun  must  be
turned  off  so  that  it does not put any  packets  on  the
Ethernet.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/

  declare


        /*  The  source address is included in the  transmit
            data block because the ni3010 is more efficient
            with  the  automatic  insertion  of  the  source
            suppressed by the "clear insert source address"
            command                                       */
     1    transmit_data_block based (trans_blk_ptr),
        2 destination_address_a   bit (8),
        2 destination_address_b   bit (8),
        2 destination_address_c   bit (8),
```

```
   2 destination_address_d    bit (8),
   2 destination_address_e    bit (8),
   2 destination_address_f    bit (8),
   2 source_address_a    bit (8),
   2 source_address_b    bit (8),
   2 source_address_c    bit(8),
   2 source_address_d    bit (8),
   2 source_address_e    bit(8),
   2 source_address_f    bit (8),
   2 type_field_a        bit(8),
   2 type_field_b        bit(8),
   2 data (100)          bit(8),


1 receive_data_block based(rec_blk_ptr),
   2 frame_status             bit (8),
   2 null_byte                bit (8) ,
   2 frame_length_lsb         bit (8) ,
   2 frame_length_msb         bit (8) ,
   2 destination_address_a    bit (8) ,
   2 destination_address_b    bit (8) ,
   2 destination_address_c    bit (8) ,
   2 destination_address_d    bit (8) ,
   2 destination_address_e    bit (8) ,
   2 destination_address_f    bit (8) ,
   2 source_address_a         bit (8) ,
   2 source_address_b         bit (8) ,
   2 source_address_c         bit (8) ,
   2 source_address_d         bit (8) ,
   2 source_address_e         bit (8) ,
   2 source_address_f         bit (8) ,
   2 type_field_a             bit (8) ,
   2 type_field_b             bit (8) ,
   2 data (1500)              bit (8) ,
   2 crc_msb                  bit (8) ,
   2 crc_upper_middle_byte    bit (8) ,
   2 crc_lower_middle_byte    bit (8) ,
   2 crc_lsb                  bit (8) ,

   /* how many packets does user want to receive */
user_desires fixed,

   /* size of the Ethernet Packet */
size_ptr pointer,
size  fixed bin(15) based (size_ptr),

   /* pointers to position overlays in memory */
(trans_blk_ptr,rec_blk_ptr) pointer,

   /* number of packets received by this program */
number_of_packets fixed static init(0),

copy_ie_register bit(8),
i  fixed bin (15),
reg_value bit (8) ,
```

```
   /*   Modules external to this module */

   add32bit   entry   (bit(8),bit(8),bit(8)),
   cksum      entry   (bit(8),fixed bin(7),bit(16)),
   write_io_port entry (bit (8) ,bit (8) ),
   read_io_port  entry (bit (8) ,bit (8) ),
   initialize_cpu_interrupts     entry,
   enable_cpu_interrupts         entry,
   disable_cpu_interrupts        entry,
   write_bar  entry (pointer);

   /*   end external module listing  */


%replace

   /* codes for programmable PIC on 86/12 SBC;
      note that icw2,icw4, and ocw are the same  */

   icw1_port_address          by 'c0'b4,
   icw2_port_address          by 'c2'b4,
   icw4_port_address          by 'c2'b4,
   ocw_port_address           by 'c2'b4,


   /* note: icw ==> initialization
                    control
                    word

            ocw ==> operational
                    command
                    word              */



   icw1                       by '13'b4,

   /* single PIC configuration, edge
      triggered input           */

   icw2                       by '40'b4,

   /* most significant bits of vectoring
   byte; for an interrupt 5,
   the effective address will be
   (icw2 + interrupt #) * 4 which
   will be (40 hex + 5) * 4 =
   114 hex              */

   icw4                     by '0f'b4,

   /* automatic end of interrupt
   and buffered mode/master   */

   ocw1                       by '9f'b4,
```

```
    /* unmask interrupt 5 (bit 5) and
       mask all others                  */

    /* end 8259a codes */


    true                        by '1'b,
    false                       by '0'b,
    vax                         by '7f'b4,
    clearscreen                 by '^z';


    /* include constants specific to the NI3010
       board and IP/TCP declarations    */

%include 'vt.dcl';
```

```
/*****************************************************************/

                        /*  Main Body */

begin;   /*  initialize the process  */

        dcl     fill_xtpa(10000) char based(trans_blk_ptr);

          /* disable the ni3010 so that it does not
             interrupt this initialization          */
          copy_ie_register = disable_ni3010_interrupts;
          call write_io_port(interrupt_enable_register,
                             disable_ni3010_interrupts);

          call read_io_port (command_status_register,
                             reg_value);
          call perform_command(go_offline);
          call perform_command(reset);

          /* clear terminal, base the transmit block and
             the receive block in the extended tpa, and
             fill the tpa with asteriks.                */
          put list ('clearscreen);
          unspec(trans_blk_ptr) = '8001'b4;
          unspec(rec_blk_ptr) = '8600'b4;
          do i = 1 to 10000;
                fill_xtpa(i) = '*';
          end;


          /* initialize the pic  */

              call write_io_port (icw1_port_address,icw1);
              call write_io_port (icw2_port_address,icw2);
              call write_io_port (icw4_port_address,icw4);
              call write_io_port (ocw_port_address,ocw1);

          /*  end initializing the pic  */

          call initialize_cpu_interrupts;
          call initialize_headers;

          call perform_command (go_online);
          call perform_command'set_promiscuous_mode);
          call perform_command(clear_insert_source_address);

          /* user interface; perform all of this with the
             board  disabled  so  that  the  i/o  runtime
             routines, which are not re-entrant, will not
             be interrupted. */
          put skip list('how many messages to receive');
          get list (user_desires);
          call make_ethernet_header;

end;    /* end initialization  */
```

```
        /* enable the board for the remote login */
        copy_ie_register = receive_block_available;
        call write_io_port (interrupt_enable_register,
                            receive_block_available);
        call remote_login;
        call perform_command (reset);

    /* end main body */


/*****************************************************************
This module is called from the main body of this program.
It initializes the IP and TCP headers, the Ethernet frame
type filed, and several other variables for use in the
remote logn. The IP header source and destination addresses
are the addresses of the VAX and Sun on the ARPANET.
*****************************************************************/

initialize_headers:             procedure;


    /*  initialize the IP header; some values are
        completely arbitrary in this program; for
        instance, the id_lsb and id_msb          */

    unspec(int_hdr_ptr)  =                  '800f'b4;
    internet_header.version_ihl =           '45'b4;
    internet_header.type_of_service =       '00'b4;
    internet_header.length_msb =            '00'b4;
    internet_header.length_lsb =            '2c'b4;
    internet_header.id_msb =                '23'b4;
    internet_header.id_lsb =                '4e'b4;
    internet_header.flags_frag =            '0000'b4;
    internet_header.ttl =                   '0f'b4;
    internet_header.protocol =              '06'b4;
    internet_header.hdr_checksum =          '0000'b4;
    internet_header.source_a =              'c0'b4;
    internet_header.source_b =              '09'b4;
    internet_header.source_c =              'c8'b4;
    internet_header.source_d =              '01'b4;
    internet_header.dest_a =                'c0'b4;
    internet_header.dest_b =                '09'b4;
    internet_header.dest_c =                'c8'b4;
    internet_header.dest_d =                '03'b4;
```

```
            /*  initialize the TCP header; some values are
                completely arbitrary; for instance, the
                sequence number, source port; NOTE: the
                source port will be stored in memory
                as 03 ff hex and the destination port
                will be stored as 02 01 hex.    */

        unspec(tcp_hdr_ptr) =              '8023'b4;
        tcp_header.source_port =           'ff03'b4;
        tcp_header.dest_port =             '0102'b4;
        tcp_header.offset =                '60'b4;
        tcp_header.control =               syn;
        tcp_header.window_msb =            '08'b4;
        tcp_header.window_lsb =            '00'b4;
        tcp_header.checksum =              '0000'b4;
        tcp_header.urgent_pointer =        '0000'b4;
        tcp_header.data(1) =               '02'b4;
        tcp_header.data(2) =               '04'b4;
        tcp_header.data(3) =               '04'b4;
        tcp_header.data(4) =               '00'b4;
        tcp_header.data(5) =               '30'b4;
        tcp_header.data(6) =               '02'b4;
        tcp_header.sequence_num(1)=        '02'b4;
        tcp_header.sequence_num(2) =       '1c'b4;
        tcp_header.sequence_num(3) =       '28'b4;
        tcp_header.sequence_num(4) =       '01'b4;


        do i = 1 to 4;
                tcp_header.ack_num(i) = '00'b4;
                inc_seq_num(i) = '00'b4;
                seg_ack(i) = '00'b4;
        end;

        seg_len = '00'b4;

        transmit_data_block.type_field_a = '08'b4;
        transmit_data_block.type_field_b = '00'b4;



end;    /*  initialize headers  */
```

```
/**************************************************************
This module is called from various other modules.  It sends
to the ni3010 board the command received in the parameter
list.  It then waits to see if the command was successfully
carried out by the board.  If the command was not
successful, an error message is displayed and the program
terminates.
**************************************************************/

perform_command:        procedure (command);

  declare
     command bit (8) ,
     reg_value bit (8) ,
     srf bit (8) ,
     write_io_port entry (bit (8) ,bit (8) ),
     read_io_port  entry (bit (8) ,bit (8) );



srf = '0'b4;
call write_io_port (command_register,command);
do while '(srf & '01'b4) = '00'b4);
        call read_io_port (interrupt_status_reg,
                           srf);
end;  /* do while */
call read_io_port (command_status_register,
                   reg_value);
if (reg_value > '01'b4) then
do;
        /* not (SUCCESS or SUCCESS with Retries) */

       put skip edit ('*** ETHERNET Board Failure ***')
                     (col(35),a);
       stop;
end; /* itd */


end perform_command;
```

```
/**********************************************************************
This module is called from the main body of the program.  It
prompts  the user for the destination of the  remote  login.
The  Ethernet  address  of  the Sun is used  as  the  source
address of the Ethernet packet.
**********************************************************************/


make_ethernet_header:          procedure;

        dcl
                choice char(1);


choice = '';

do while (choice ^= 'a' & choice ^= 'b' &
          choice ^= 'c' & choice ^= 'd');
  put skip list
('Chose from below menu the destination of your packet');
  put skip;
  put skip list
('                    NOTE: enter only lower case letters');
  put skip;
  put skip edit
('     a.  VAX UNIX','     b.  SUN1')(a,skip,a,skip);
  put skip edit
('     c.  MYSELF','     d.  MDS')(a,skip,a);
  put skip;
  get edit(choice)(a(1));
  put skip;
end;  /* do while */


if (choice = 'a' | choice = 'c' | choice = 'd') then
  do;
     transmit_data_block.destination_address_a = '02'b4;
     transmit_data_block.destination_address_b = '07'b4;
     transmit_data_block.destination_address_c = '01'b4;
     transmit_data_block.destination_address_d = '00'b4;
  end;
else
  do;
     transmit_data_block.destination_address_a = '02'b4;
     transmit_data_block.destination_address_b = '60'b4;
     transmit_data_block.destination_address_c = '8c'b4;
     transmit_data_block.destination_address_d = '00'b4;
     transmit_data_block.destination_address_e = '42'b4;
     transmit_data_block.destination_address_f = '27'b4;
   end;
```

```
if choice = 'a' then
  do;
     transmit_data_block.destination_address_e = '07'b4;
     transmit_data_block.destination_address_f = '7f'b4;
  end;

else if choice = 'c' then
  do;
     transmit_data_block.destination_address_e = '03'b4;
     transmit_data_block.destination_address_f = 'ea'b4;
  end;
else if choice = 'd' then
  do;
     transmit_data_block.destination_address_e = '04'b4;
     transmit_data_block.destination_address_f = '0a'b4;
  end;

transmit_data_block.source_address_a = '02'b4;
transmit_data_block.source_address_b = '60'b4;
transmit_data_block.source_address_c = '8c'b4;
transmit_data_block.source_address_d = '00'b4;
transmit_data_block.source_address_e = '42'b4;
transmit_data_block.source_address_f = '27'b4;



end;  /* make_ethernet_header */


/*****************************************************************
This module is called from the remote_login module. The
parameter is the number of bytes in the data portion of the
packet to be transmitted, i.e. the number of bytes in the
IP/TCP protocols. The parameter is incremented by 14, the
number of bytes in the Ethernet header of the transmit data
block. Should the clear insert physical address mode
change, then it should be incremented by 8 vice 14. The
parameter passed is a fixed bin(7); it must be overlayed
with a bit(8) variable so that it can be written out to the
board's low byte count register.
*****************************************************************/

transmit_packet:          procedure (low_byt_cnt);

  declare
     write_io_port entry (bit (8) ,bit (8) ),
     read_io_port entry  (bit (8) ,bit (8) ),
     enable_cpu_interrupts          entry,

        /* used to overlay the fixed bin(7) parameter */
     cnt bit(8) based(byt_cnt_ptr),
     byt_cnt_ptr         pointer,
     disable_cpu_interrupts         entry,
     low_byt_cnt fixed bin(7),
```

```
     write_bar entry (pointer);

call disable_cpu_interrupts;

do while (copy_ie_register = transmit_dma_done
                           |
          copy_ie_register = receive_dma_done);

    call enable_cpu_interrupts;
    do while (copy_ie_register = transmit_dma_done
                               |
                copy_ie_register = receive_dma_done);
    end;
    call disable_cpu_interrupts;
end;

copy_ie_register = disable_ni3010_interrupts;
call write_io_port(interrupt_enable_register,
                   disable_ni3010_interrupts);

call enable_cpu_interrupts;
call write_bar (addr(transmit_data_block));
call write_io_port(high_byte_count_reg,'00'b4);

  /* increment the parameter and overlay it */
low_byt_cnt = low_byt_cnt + 14;
byt_cnt_ptr = addr(low_byt_cnt);
call write_io_port(low_byte_count_reg,cnt);

call disable_cpu_interrupts;
copy_ie_register = transmit_dma_done;
call write_io_port(interrupt_enable_register,
                   transmit_dma_done);
call enable_cpu_interrupts;

do while (copy_ie_register = transmit_dma_done);
end;   /* loop until the interrupt handler
          takes care of the TDD interrupt -
          it sets IE_REG to 4 */

call perform_command(load_and_send);


end transmit_packet;
```

```
/*************************************************************/
This module is called by the assembly language interrupt
routine.  It handles three ni3010 interrupts: the receive
block available, receive dma done, and the transmit dma
done interrupts.  It also checks for the ni3010 size
anomally; if found, it prints out an error messsage and
terminates the program.  If a packet is received from the
VAX, the variable "seg_ack" is updated so that the outgoing
packet's acknowledgement number can be updated in the
remote_login module.  The variable "size" is overlaid on the
frame length of the ethernet packet; it is the length from
the first destination byte to last CRC byte; it is used to
reposition the receive_data_block in memory so that no
portion of th previous packet is overwritten.  When
examining the packets in memory with DDT-86, each will be
separaed by the asteriks that were used to fill a large
portion of the extended tpa.
*************************************************************/

HL_interrupt_handler:          procedure external;


  declare
     source bit(8) external,
     write_io_port entry (bit (8) ,bit (8) ),
     read_io_port entry (bit (8) , bit(8) ),
     enable_cpu_interrupts          entry,
     user_desires fixed external,
     disable_cpu_interrupts         entry,
     write_bar entry (pointer);


call disable_cpu_interrupts;
call write_io_port(interrupt_enable_register,
                   disable_ni3010_interrupts);
if (copy_ie_register = receive_block_available)
then do;
        /* gives address to board for next dma
           of packet into memory                 */
     call write_bar(addr(receive_data_block));
     call write_io_port(high_byte_count_reg,'06'b4);
     call write_io_port(low_byte_count_reg,'00'b4);

     /* initiate receive DMA */

     copy_ie_register = receive_dma_done;
     call write_io_port(interrupt_enable_register,
                        receive_dma_done);

end;   /* if receive=block_available */
```

```
else

     if (copy_ie_register = receive_dma_done) then
     do;
        number_of_packets=number_of_packets + 1;

        size_ptr= addr(receive_data_block.frame_length_lsb);
        if (size > 1522 | size < 64) then
           do;
              put skip list('SIZE ERROR');
              stop;
           end;

        source = '00'b4;
        If  (receive_data_block.source_address_f = vax) then
           do;
              source = vax;
              do i = 1 to 4;
                 seg_ack(i) =receive_data_block.data(i + 24);
              end;
           end;

        rec_blk_ptr = addr(receive_data_block.data(size-4));
        copy_ie_register = receive_block_available;
        call write_io_port(interrupt_enable_register,
                           receive_block_available);

     end;  /* if receive dma done */

else

  if (copy_ie_register = transmit_dma_done)
     then do;
        copy_ie_register = receive_block_available;
        call write_io_port(interrupt_enable_register,
                           receive_block_available);

     end;   /* if transmit dma done */


end HL_interrupt_handler;
```

```
/*****************************************************************/
This module is called from the remote_login module.  It
calculates the checksum for the TCP header.  Since the
calculation involves a pseudo header prepended to the TCP
header, the variable "temp" stores all the bytes of memory
located just prior to the TCP header (that it, the last 12
bytes of the IP header).  Overlays are used that bit(8)
values can be used as integer values without suffering from
conversion factors.  The variable "int_length" is the length
of the IP dtatgram and therefore includes the TCP segment.
The variable "seg_len" is the length of the TCP segment.  If
the length of the segment is an odd number, then an extra
null byte is added so that there are full 16 bit words for
the checksum calculation.  NOTE: the divide function is
used;  it returns the floor value should there be a
remainder from the division.
*****************************************************************/

calc_tcp_cksum:                    procedure;


  dcl
        cksum   entry (bit(8),fixed bin(7),bit(16)),
        fixed_overlay   bit(8) based(fixed_overlay_ptr),
        fixed_overlay_ptr   pointer,
        int_length   fixed bin(7) based(int_length_ptr),
        int_length_ptr   pointer,
        num16bit_wds     fixed bin(7),
        odd   bit(1),  /* odd IP length */
        temp(13)   bit(8);



odd = false;
do i = 1 to 12;  /* save last 12 bytes of IP header */
        temp(i) = transmit_data_block.data(i + 8);
end;

/* form tcp PSEUDO HEADER */

transmit_data_block.data(9)  = 'c0'b4;
transmit_data_block.data(10) = '09'b4;
transmit_data_block.data(11) = 'c8'b4;
transmit_data_block.data(12) = '01'b4;
transmit_data_block.data(13) = 'c0'b4;
transmit_data_block.data(14) = '09'b4;
transmit_data_block.data(15) = 'c8'b4;
transmit_data_block.data(16) = '03'b4;
transmit_data_block.data(17) = '00'b4;
transmit_data_block.data(18) = '06'b4;
transmit_data_block.data(19) = '20'b4;
```

```
  /* overlay bit(8) with fixed value */
int_length_ptr  = addr(internet_header.length_lsb);
seg_len = (int_length - 22);
num16bit_wds = divide((int_length - 8),2,7);

  /* check if length_is odd number of bytes */
if ( mod(seg_len,2)  = 0 ) then
  do;
     odd = true;
     num16bit_wds = num16bit_wds + 1;
     temp(13) = transmit_data_block.data(int_length +1);
     transmit_data_block.data(int_length +1) = '00'b4;
  end;

  /* insert byte 12 of pseudo header */
fixed_overlay_ptr = addr(seg_len);
transmit_data_block.data(20) = fixed_overlay;

call cksum(transmit_data_block.data(9),
           num16bit_wds,tcp_header.checksum);

  /* restore IP */
do i = 1 to 12;
        transmit_data_block.data(i + 8) = temp(i);
end;
if odd = true then
  transmit_data_block.data(int_length +1) = temp(13);



end calc_tcp_cksum;



/*****************************************************************
This  module sends the packets necessary for the  connection
estalishment  to the VAX and the remote login protocol up to
the  point where the VAX asks for "Password".  Each  packet
update  changes only those portions of the packet  that  are
different  from  one  packet  to  the  next.  There  is  no
processing  of  inbound  packets  except  for  VAX  seauence
numbers and data to be pushed.  It assumes no deviation from
a normal login seaunece.
*****************************************************************/

remote_login:       procedure;

  declare
     source bit(8) external,
     packet_length fixed bin(7) external,
     add32bit entry (bit(8),bit(8),bit(8)),
     cksum entry (bit(8),fixed bin(7),bit(16)),
     write_io_port entry (bit (8) ,bit (8) ),
     num16bit_wds fixed bin(7) external;
```

```
/*  send packet 1 */
num16bit_wds = 10;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
call calc_tcp_cksum;
packet_length = 46;
call transmit_packet(packet_length);


/* assume VAX will acknowledge it #1, wait until it does */
do while ( source ^= vax );
end;



/*  packet_2 update  */

inc_seq_num(4) = '01'b4;
call add32bit(tcp_header.sequence_num(4),inc_seq_num(4),
        tcp_header.sequence_num(4));
tcp_header.offset = '50'b4;
tcp_header.control = ack;
call add32bit(seg_ack(4),inc_seq_num(4),tcp_header.ack_num(4));
internet_header.length_lsb = '28'b4;
internet_header.id_lsb = '02'b4;


/*  end packet_2 update  */

num16bit_wds = 10;
internet_header.hdr_checksum= '0000'b4;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
tcp_header.checksum = '0000'b4;
call calc_tcp_cksum;
packet_length = 46;
call transmit_packet(packet_length);


/* packet 3 update */

internet_header.length_lsb = '29'b4;
tcp_header.control = ack_psh;
tcp_header.data(1) = '00'b4;
tcp_header.data(2) = '04'b4;
tcp_header.data(3) = '04'b4;
tcp_header.data(4) = '00'b4;
tcp_header.data(5) = '00'b4;
tcp_header.data(6) = '00'b4;
```

```
internet_header.id_lsb = '03'b4;
num16bit_wds = 10;
internet_header.hdr_checksum = '0000'b4;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
tcp_header.checksum = '0000'b4;
call calc_tcp_cksum;
packet_length = 46;
call transmit_packet(packet_length);


/* packet 4 update */

internet_header.length_lsb = '2e'b4;
inc_seq_num(4)= '01'b4;
call add32bit(tcp_header.sequence_num(4),inc_seq_num(4),
                tcp_header.sequence_num(4));

tcp_header.data(1) = '72'b4;
tcp_header.data(2) = '65'b4;
tcp_header.data(3) = '65'b4;
tcp_header.data(4) = '6b'b4;
tcp_header.data(5) = '65'b4;
tcp_header.data(6) = '00'b4;

internet_header.id_lsb = '04'b4;
num16bit_wds = 10;
internet_header.hdr_checksum = '0000'b4;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
tcp_header.checksum = '0000'b4;
call calc_tcp_cksum;
packet_length = 46;
call transmit_packet(packet_length);


/* segment 5 update */

inc_seq_num(4)= '06'b4;
call add32bit(tcp_header.sequence_num(4),inc_seq_num(4),
                tcp_header.sequence_num(4));
internet_header.id_lsb = '05'b4;
num16bit_wds = 10;
internet_header.hdr_checksum = '0000'b4;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
tcp_header.checksum = '0000'b4;
call calc_tcp_cksum;
packet_length = 46;
call transmit_packet(packet_length);
```

```
/* packet 6 update */

internet_header.length_lsb ='31'b4;
inc_seq_num(4)='06'b4;
call add32bit(tcp_header.sequence_num(4),inc_seq_num(4),
                tcp_header.sequence_num(4));
internet_header.id_lsb ='06'b4;

tcp_header.data(1) ='73'b4;
tcp_header.data(2) ='75'b4;
tcp_header.data(3) ='6e'b4;
tcp_header.data(4) ='2f'b4;
tcp_header.data(5) ='39'b4;
tcp_header.data(6) ='36'b4;
tcp_header.data(7) ='30'b4;
tcp_header.data(8) ='30'b4;
tcp_header.data(9) ='00'b4;

num16bit_wds = 10;
internet_header.hdr_checksum = '0000'b4;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
tcp_header.checksum = '0000'b4;
call calc_tcp_cksum;
packet_length = 49;
call transmit_packet(packet_length);


/* wait unti vax replies */

do while (source ^= vax);
end;


/* wait until vax replies with its null byte */

do while  (  (source = vax) &
          (receive_data_block.data(34) ^= ack_psh) );
end;


/* packet 7 update */

internet_header.length_lsb = '28'b4;
inc_seq_num(4)='09'b4;
call add32bit(tcp_header.sequence_num(4),inc_seq_num(4),
                tcp_header.sequence_num(4));
inc_seq_num(4)='01'b4;
call add32bit(tcp_header.ack_num(4),inc_seq_num(4),
                tcp_header.ack_num(4));
tcp_header.control = ack;
internet_header.id_lsb ='07'b4;
```

```
tcp_header.data(1) ='2f'b4;
tcp_header.data(2) ='39'b4;
tcp_header.data(3) ='36'b4;
tcp_header.data(4) ='30'b4;
tcp_header.data(5) ='30'b4;
tcp_header.data(6) ='00'b4;

num16bit_wds = 10;
internet_header.hdr_checksum = '0000'b4;
call cksum(internet_header.version_ihl,num16bit_wds,
                internet_header.hdr_checksum);
tcp_header.checksum = '0000'b4;
call calc_tcp_cksum;
packet_length = 46;
call transmit_packet(packet_length);



put skip list ('PACKET #7 SENT');
put skip;
do while (number_of_packets < user_desires );
end; /* do while */


end remote_login;

/*****************************************************************/


end;
```

APPENDIX E

SOURCE CODE FOR 32 BIT ADDITION

```
;EXAMPLE OF CALL TO THIS PROCEDURE
;call add32bit(number1(4),number2(4),result(4)) where:
;
;       parameter 1 is the least significant byte of a
;         32bit number (array of 4 bytes)
;       parameter 2 is the lsb of another 32 bit number
;       parameter 3 is the lsb of the result
;
;
;The 32 bit numbers are stored in memory such that the high
;order byte occupies the lowest memory location.  This is
;contrary to the way that PL1 stores numbers.  The numbers
;are stored in this manner because they have been DMA'd
;into memory by the NI3010 board in the order they were
;recieved over the Ethernet, i.e., high order byte
;recieved first.
;
;The algorithm adds the numbers byte by byte starting with
;the low order bytes and places the rseultant byte addition
;into the corresponding result byte position.




public  add32bit


add32bit:



cseg

push ax! push bx! push cx! push dx! push si! push di!

mov     cx,04h          ;number of bytes to be added
                        ;(32 bit number)
mov     si,[bx]         ;load index registers with address
mov     di,2[bx]        ;the lsb of each number to be added
mov     dx,4[bx]        ;pointer to the lsb of the result

clc
```

```
beg:
mov     al,[si]         ;byte of first number to add
mov     ah,[di]         ;byte of other number to add
adc     ah,al
dec     si              ;decrement pointer and store it away
push    si
mov     si,dx           ;make si now point to result
mov     [si],ah         ;move addintion into result

dec     si              ;step down in memory
mov     dx,si           ;save pointer to result
pop     si
dec     di
loop    beg


pop di! pop si! pop dx! pop cx! pop bx! pop ax!

ret
```

APPENDIX F

CHECKSUM PROGRAMMING NOTES AND SOURCE CODE

The checksum algorithm for the IP header and the TCP segment are the same. The official specification for the algorithm is:

> The Checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words; for purposes of computing the checksum, the checksum field is zero.[Ref. 3]

Another source lists the algorithm as:

> The checksum field is to simply add up all the data regarded as 16 bit words, and then to take the one's complement of the sum.[Ref. 6]

The assembly language program below always agreed with the checksum calculations that were contained in the packets. Initial attempts yielded inconsistent reults when compared to those in the UNIX produced packets. Some attempts always produced the correct result for the IP header and 90% of the time for the TCP header. Other attempts were less consistant than the above example. Since one's complement arithmetic is used, the erroneous implementations can produce results that are quite close to those produced by the UNIX operating system.

The program below implements the algorithm as if the 16 bit words were integers stored with the least significant byte in low memory, followed by the most significant byte. Note that the first byte of the word is loaded into the low

side of the 16 bit registers in the 8086 micro-processor and the second byte loaded into the high side of the registers.

```
;EXAMPLE OF CALL TO THIS PROCEDURE
;call cksum(x,y,z) where:
;
;       parameter1 (bit (8))  is the starting byte of data
;                             to be summed
;       parameter2 (fixed bin 7) is one-half the length of
;                             the block (# of 16 bit words)
;       parameter3 (bit(16))  the resultant 16 bit checksum
;
;The dx register will contain the resultant checksum.


public  cksum


cksum:


cseg


push ax! push bx! push cx! push dx! push si!

mov     si,2[bx]        ;load cx with one-half the length
mov     cx,[si]
mov     ch,00h          ;zero out high side of cx register
mov     dx,0000h        ;zero out running total of checksum

mov     si,[bx]         ;si points to first byte of block
clc


again:
mov     al,[si]         ;msb of 16 bit word
inc     si
mov     ah,[si]         ;lsb of 16 bit word
adc     dx,ax           ;add to running total
inc     si
loop    again

xor     dx,0ffffh       ;1's complement of running total

mov     si,4[bx]        ;place result in return parameter
mov     [si],dl
inc     si
```

```
mov     [si],dh

pop  si! pop  dx! pop  cx! pop  bx! pop  ax!
ret
```

APPENDIX G

INCLUDE FILE FOR PROGRAM LISTEN

```
%replace

  /*                  I/O port addresses

    These values are specific to the use of the INTERLAN
    NI3010 MULTIBUS to ETHERNET interface board. Any
    change to the I/O port address of '00b0' hex (done
    so with a DIP switch) will require a change to
    these addresses to reflect that change.

                                                        */


          command_register                by 'b0'b4,
          command_status_register         by 'b1'b4,
          transmit_data_register          by 'b2'b4,
          interrupt_status_reg            by 'b5'b4,
          interrupt_enable_register       by 'b8'b4,
          high_byte_count_reg             by 'bc'b4,
          low_byte_count_reg              by 'bd'b4,


      /*  end of I/O port addresses */

      /*  Interrupt enable status register values */
           disable_ni3010_interrupts      by '00'b4,
           ni3010_intrpts_disabled        by '00'b4,
           receive_block_available        by '04'b4,
           transmit_dma_done              by '06'b4,
           receive_dma_done               by '07'b4,


      /*   end register values */

      /*   Command Function Codes */

           module_interface_loopback      by '01'b4,
           internal_loopback              by '02'b4,
           clear_loopback                 by '03'b4,
           go_offline                     by '08'b4,
           go_online                      by '09'b4,
           onboard_diagnostic             by '0a'b4,
           load_transmit_data             by '28'b4,
```

```
        load_and_send                   by '29'b4,
        reset                           by '3f'b4,
        set_promiscuous_mode            by '04'b4,
        clear_insert_source_address     by '0e'b4;

/*      end Command Function Codes        */
```

APPENDIX H

INCLUDE FILE FOR PROGRAM DEMO

```
declare
        1  internet_header based(int_hdr_ptr),
          2  version_ihl          bit(8),
          2  type_of_service      bit(8),
          2  length_msb           bit(8),
          2  length_lsb           bit(8),
          2  id_msb               bit(8),
          2  id_lsb               bit(8),
          2  flags_frag           bit(16),
          2  ttl                  bit(8),
          2  protocol             bit(8),
          2  hdr_checksum         bit(16),
          2  source_a             bit(8),
          2  source_b             bit(8),
          2  source_c             bit(8),
          2  source_d             bit(8),
          2  dest_a               bit(8),
          2  dest_b               bit(8),
          2  dest_c               bit(8),
          2  dest_d               bit(8),


        1  tcp_header based(tcp_hdr_ptr),
          2  source_port          bit(16),
          2  dest_port            bit(16),
          2  sequence_num(4)      bit(8),
          2  ack_num(4)           bit(8),
          2  offset               bit(8),
          2  control              bit(8),
          2  window_msb           bit(8),
          2  window_lsb           bit(8),
          2  checksum             bit(16),
          2  urgent_pointer       bit(16),
          2  data(512)            bit(8),


        int_hdr_ptr      pointer,
        tcp_hdr_ptr      pointer,
        seg_len          fixed bin(7),
```

```
/*  These 4 element arrays are 32 bit variables   */

            inc_seq_num(4)  bit(8),
            seg_ack(4)      bit(8);

     %replace

          /*  Control bit for the TCP header  */

              ack             by '00010000'b,
              ack_psh         by '00011000'b,
              syn             by '00000010'b,
              ack_fin         by '00010001'b,
              ack_syn         by '00010010'b,



/*                I/O port addresses

These values are specific to the use of the INTERLAN  NI3010
MULTIBUS to ETHERNET interface board.  Any change to the I/O
port  address of '00b0' hex (done so with a DIP switch) will
require a change to these addresses to        reflect  that
change.

                                                          */


              command_register              by 'b0'b4,
              command_status_register       by 'b1'b4,
              transmit_data_register        by 'b2'b4,
              interrupt_status_reg          by 'b5'b4,
              interrupt_enable_register     by 'b8'b4,
              high_byte_count_reg           by 'bc'b4,
              low_byte_count_reg            by 'bd'b4,

          /*  end of I/O port aidresses */


          /*  Interrupt enable status register values */

               disable_ni3010_interrupts    by '00'b4,
               ni3010_intrpts_disabled      by '00'b4,
               receive_block_available      by '04'b4,
               transmit_dma_done            by '06'b4,
               receive_dma_done             by '07'b4,

          /*   end register values */
```

```
/*   Command Function Codes */

     module_interface_loopback      by '01'b4,
     internal_loopback              by '02'b4,
     clear_loopback                 by '03'b4,
     go_offline                     by '08'b4,
     go_online                      by '09'b4,
     onboard_diagnostic             by '0a'b4,
     load_transmit_data             by '28'b4,
     load_and_send                  by '29'b4,
     reset                          by '3f'b4,
     set_promiscuous_mode           by '04'b4,
     clear_insert_source_address    by '0e'b4;

/*   end Command Function Codes     */
```

LIST OF REFERENCES

1. Stotzer, M. D., A Layered Communication System for Ethernet, M. S. Thesis, Naval Postrgraduate School, Monterey, California, September, 1983.

2. Netniyom, T. P., Design and Implementation of Software Protocols in VAX/VMS Using Ethernet Local Area Network, M. S. Thesis, Naval Postgraduate School, Monterey, California, June, 1983.

3. Network Information Center, Internet Protocol Transition Workbook, SPI International, 1982.

4. Universtiy of California, Berekley, Department of Electrical Engineering and Computer Science, UNIX Programmer's Manual, 1983.

5. Brewer. D. J., A Real Time Executive for Multiple Computer Networks, M. S. Thesis, Naval Postgraduate School, Monterey, California, December, 1984.

6. Tannenbaum, A. S., Computer Networkks, Prentice Hall, 1981.

BIBLIOGRAPHY

Digital Research, Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems, 1982.

Digital Research, PL/I Programmers Guide, 1982.

Digital Research, PL/I Reference Manual, 1982.

Intel Coorporation, MCS-80 Macro Assembly Language Reference Manual, 1979.

Intel Corporation, SBC 86/12 Single Board Computer Hardware Reference Manual, 1979.

Interlan Incorporated, NI3010 Ethernet Communications Controller User Manual, 1982.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Professor Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93943	3
5.	Professor Roger Marshall, Code 52 Department of Computer Scinece Naval Postgraduate School Montery, California 93943	1
6.	LtCol. David Melchar, USMC United States Marine Corps Representative Naval Postgraduate School Monterey, California 93943	1
7.	LtCol. John D. Reeke, USMC 9547 University Avenue Des Moines, Iowa 50322	1
8.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943	1
9.	Lt. David J. Brewer, USN 7659 Cherrywood Drive Charleston Heights, South Carolina 29405	1

Thesis
R2737 Reeke
c.1 Remote terminal login from a microcomputer to the UNIX operating system using Ethernet as the communications medium.